Converting Unstructured into Semi-Structured Process Models

Rik Eshuis\textsuperscript{a}, Akhil Kumar\textsuperscript{b}

\textsuperscript{a}Eindhoven University of Technology, School of Industrial Engineering
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
\textsuperscript{b}Department of Supply Chain and Information Systems, Smeal College of Business, Penn State University, University Park, PA 16802, USA

Abstract

Business process models capture process requirements that are typically expressed in unstructured, directed graphs that specify parallelism. However, modeling guidelines or requirements from execution engines may require that processes models are structured in blocks. The goal of this paper is to define an automated method to convert an unstructured process model containing parallelism into an equivalent semi-structured process model, which contains blocks and synchronization links between parallel branches. We define the method by means of an algorithm that is based on dominators, a well-known technique from compiler theory for structuring sequential flow graphs. The method runs in polynomial time. We implemented and evaluated the algorithm extensively. In addition we compared the method in detail with the BPStruct method from literature. The comparison shows that our method can handle cases that BPStruct is not able to and that the method coincides with BPStruct for the cases that BPStruct is able to handle.

Keywords: Business process management, process models, semi-structured processes, model transformation.

1. Introduction

Business processes management (BPM) \cite{1, 2} focuses on the automation of business processes using middleware information technology. Business process models play a key role in BPM. They can be conceptual-level designs of business processes, but they can also specify the coordination logic for execution engines that support the operational management of business processes. Communication of business process models between different stakeholders, end users or system engineers is of utmost importance to realize a successful implementation.

An important subclass of process models are structured process models, which consist of nested blocks and each block has a single entry point and
a single exit point. Examples of structured languages are the Business Process Execution Language (BPEL) [3] and OWL-S [4], a language for describing semantic web services using the Web Ontology Language (OWL). Structured process models are also required by advanced BPM techniques such as process views in inter-organizational collaborations [5, 6], rule-driven implementations of process models [7], and dynamic changes of running processes [8].

A structured process model is similar to a (parallel) program without goto statements. While every unstructured program has a structured equivalent [9], this is not true for unstructured process models, since synchronization links between parallel blocks cannot be expressed in a structured process model [10, 11]. Therefore, established BPM languages like BPEL [3] and Adept [8] allow semi-structured process models, which are structured into blocks with additional synchronization links between parallel blocks.

Structured process models are easier to read and understand than unstructured models [12] and contain fewer errors [13]. Structuredness has been proposed as modeling guideline [14]. But business processes are typically modeled in graph-based notations such as BPMN [15] or UML activity diagrams [16] that do not syntactically enforce structuredness. A graphical representation enables easy communication with end-users in the organization in which the process is used. Nodes are either activities (tasks) or routing constructs like a choice split or a parallel join while edges represent ordering constraints.

In a structured process graph, each split has a matching join, and the split and join demarcate a subprocess that corresponds to a single-entry, single-exit block. Since a split does not need to have a matching join, process graphs are usually not structured into blocks. Therefore translations have been proposed to structure graph-based process models [11, 17, 18]. But these approaches do no construct semi-structured process models.

We aim to define an automated method for converting an unstructured process model into an equivalent semi-structured process model. The method is described as a formal algorithm, using well-known concepts from the field of compiler design [19] for structuring control flow graphs. However, a control flow graph only specifies sequential behavior while a process model also specifies parallel behavior, which complicates the structuring procedure. Another difference is that the approach allows the identification of semi-structured process models in which parallel branches use cross-synchronization.

For every input process model, the method delivers output, but the result is only meaningful (i.e., the output process is equivalent to the input process) if the input process model is correct, i.e., it is free of deadlocks and there is no lack of synchronization (see Section 3). Moreover, the approach only outputs a semi-structured process model if no equivalent structured process model exists.

Compared to existing approaches for structuring unstructured process models [11, 17, 18], our method is more powerful, since it can generate both structured and semi-structured processes, while existing approach can only generate structured processes and fail for correct process models that have no equivalent structured process. The overall time complexity of the method is polynomial, which ensures that the method scales well to large process models. Other ap-
approaches are either as efficient but less powerful [11, 17] or have a much higher time complexity [18]. An extensive discussion of related work can be found in Section 7.

To simplify the presentation, we restrict ourselves in the main text to acyclic process models. In the Appendix, we extend the algorithms of the main text to deal with cyclic process models in which each loop has a single entry and a single exit point. Other papers [20, 21, 22] already discuss techniques to turn a process model with unstructured loops into an equivalent process model in which each loop has a single entry and a single exit point. These techniques are complementary to our method and can be easily integrated.

The rest of this paper is organized as follows. Section 2 presents a motivating example. Section 3 defines process flow graphs, which formalize business process models, and a functional notation for representing semi-structured processes. To simplify the definition of the structuring algorithm, we apply in Section 4 preprocessing steps to the input process flow graphs. Section 5 defines the structuring algorithm that transforms a process flow graph into an equivalent semi-structured process. The correctness of the algorithm is also analyzed in Section 5. To simplify the exposition, we consider acyclic process flow graphs in Section 4 and 5. Appendix A explains how the algorithm extends to process flow graphs with loops. Section 6 discusses evaluation of the method. To evaluate feasibility, we have implemented a prototype implementation of the method. To evaluate the utility, we have applied the method to several benchmark examples taken from the literature. Section 7 discusses related work. Finally, Section 8 concludes this paper and gives directions for further work.

2. Motivating example

We motivate the approach with an example process model shown in Figure 1. The notation is explained in the next section. The process is about handling insurance claims for damaged vehicles. Each received claim is assessed. For small amounts, the claim is accepted without further ado. For large payment amounts, the vehicle is inspected and the payment is determined and approved by a manager. In both cases, the claim is paid and in parallel a decision letter is sent, where the decision is subject to the manager approval for large amounts. In parallel, each assessed claim is monitored and a report is created. Finally, the claim is archived if all previous tasks have completed.

The resulting process flow graph in Figure 1 cannot be converted into a structured process since there is a cross-synchronization between two parallel branches (arrow from AND split A3 to AND join A4). Therefore, all existing approaches for structuring process flow graphs [11, 17, 18] fail for this example.

The approach defined in this paper converts the process flow graph into the semi-structured process shown in Figure 2. The synchronization constraint between A3 and A4 in Figure 1 is translated into condition done(A3) for task File report in Figure 2, where A3 is a dummy task whose sole purpose is to ensure that File report starts at the proper moment. In the remainder of this paper, we revisit this example to explain and highlight key elements of the approach.
3. Preliminaries

We define semi-structured process flow graphs. We also introduce definitions based on dominators from compiler theory [19] to identify structure in process flow graphs. These definitions are used in the structuring algorithm presented in the next section.

3.1. Process flow graphs

We define a process flow graph as a generalization of a program flow graph [19]. The main difference is that process flow graphs allow parallelism, while program flow graphs are strictly sequential.

A process flow graph $\mathcal{P}$ is a tuple $(N, T, A, X, \text{start}, \text{end}, E, \text{synch})$ where

- $N$ is a set of nodes, partitioned in sets $T$, $A$, $X$, and $\{\text{start}, \text{end}\}$.
- $T$ is a set of tasks,
- $A$ is a set of AND connector nodes,
- $X$ is a set of XOR connector nodes,
– start is the unique start node,

– end is the unique end node,

– \( E \subseteq N \times N \) is the control flow (transition) relation,

– \( \text{synch} : N \to BExp \) is a partial function that maps nodes to synchronization conditions, which are boolean expressions of set \( BExp \). In the figures, each synchronization condition is demarcated with square brackets. Synchronization conditions only exist for semi-structured processes, defined below, not for unstructured processes.

For process flow graphs we use some standard definitions from graph theory [23]. If \((n_1, n_2) \in E\), then \( n_1 \) is predecessor of \( n_2 \) and \( n_2 \) is successor of \( n_1 \).

A directed path is a sequence of nodes \( n_1 n_2 .. n_l \) such that for every \( i, 0 \leq i < l, (n_i, n_{i+1}) \in E \).

We use the following syntactic constraints on process flow graphs. As usual for program flow graphs, we require that node start has no incoming edges, node end no outgoing edges, and each node in \( N \) lies on a directed path from start to end. Moreover, each task node should have exactly one incoming and one outgoing edge; start has one outgoing edge but no incoming edge; and end has one incoming edge but no outgoing edge.

Next, we identify several other kinds of nodes in process flow graphs. A node with more than one outgoing edge is called a split node. A node with more than one incoming edge is called a join node. Since we require that the start, end and task nodes have at most one incoming and one outgoing edge, each split and each join node is always an AND or a XOR connector node. To simplify the presentation, we require that a split node is not also a join node. For technical reasons, we do allow that an AND or XOR node is neither a split nor a join, in which case it has one incoming and one outgoing edge.

**Semi-structured process flow graphs**

Structured process graphs are graphs with the property that the (AND and XOR) control nodes occur in (split, join) pairs, and these pairs are properly nested inside each other [11]. Each pair demarcates a block with a single entry and a single exit point. For such graphs we can represent blocks in a simple functional notation using \( A \) and \( X \) as pairs of AND and XOR splits respectively, and ‘;’ as the sequence operator. Semi-structured process graphs are structured process graphs that use synchronization conditions on tasks, specified with function \( \text{synch} \). The synchronization condition of a task restricts when the task is allowed to start. Synchronization conditions are expressed in square brackets. Thus, the semi-structured process of Figure 2 is expressed in our notation as:

\begin{verbatim}
Start; Assess claim; P(Monitor claim; File report [done(A3)], X(Inspect vehicle; A3; P(Send decision, Approve payment; Pay), Accept claim; A3; P(Send decision, Pay))); Activate; End
\end{verbatim}
Notice how the synchronization condition done(A3) appears next to File report task indicating that it is a precondition for this task. This notation is convenient for textual representation. Given a process flow graph in this notation it is straightforward to convert it to its visual representation.

**Visualization**

We visualize process flow graphs using the Business Process Modeling Notation (BPMN)\[15\]. Rectangles represent tasks, a circle represents start, a circle with a bold border represents end, AND nodes are represented by diamonds with a ‘+’, and XOR nodes by diamonds with an ‘X’.

**Correctness**

A process flow graph can be incorrect in the sense that during execution it may get stuck. Sadiq and Orlowska [24] have identified two correctness properties for process models that apply directly to process flow graphs as well. They use the auxiliary notation of an instance subgraph, which is a maximal subgraph such that for each XOR node in the subgraph, one predecessor and one successor is also in the subgraph, and for each AND node all predecessors and successors are in the subgraph. An instance subgraph contains a deadlock if it contains some predecessor nodes of an AND node but not all. An instance subgraph has lack of synchronization if it contains more than one predecessor node of a XOR join.

A process flow graph is correct if each instance subgraph derived from it has no deadlock and no lack of synchronization. More elaborate and precise definitions can be found elsewhere [24, 25]. These citations also contain efficient approaches for verifying correctness. Structured process models are by construction correct. For the remainder of this paper, we only consider correct process flow graphs.

### 3.2. Identifying structure in process flow graphs

We introduce mathematical notions on process flow graphs that help to define the structuring algorithm in Section 5. The notions build on the well-known concept of a dominator [19].

For a process flow graph \((N, T, A, X, start, end, E, grd)\), node \(p\) dominates another node \(q \neq p\) if every path from \(start\) to \(q\) passes through \(p\). Node \(p\) is the immediate dominator of node \(q\), if \(p \neq q\) and every dominator of \(q\) other than \(p\) also dominates \(p\). Let \(IDOM(q)\) denote the immediate dominator of \(q\). Symmetrically, node \(p\) post-dominates a node \(q \neq p\), if every path from \(q\) to \(end\) passes through \(p\). Let \(IPDOM(q)\) denote the unique immediate post-dominator of \(q\); so every post-dominator of \(q\) other than \(p\) also post-dominates \(IPDOM(q)\). The immediate dominator and immediate post-dominator are unique [19].

**Example 1.** Figure 3 shows the dominators for Figure 1 in a dominator tree [19]. In the tree, each parent of a child is the immediate dominator of that child. In a similar way, post-dominators can be represented in a tree. For instance, the immediate post-dominator of \(A_1\) is \(A_3\). □
We use IDOM and IPDOM to identify for each node $n$ a set $FOLLOW(n)$ containing its successor nodes at the same level of nesting in the structured process model. For each node $n_1, n_2 \in N$, let

$$FOLLOW(n_1) = \{ n_2 \mid n_1 = IDOM(n_2) \land n_2 = IPDOM(n_1) \}.$$ 

If $n_1$ is a split node and $n_1 = IDOM(n_2)$, node $n_2$ must have two or more incoming edges, hence it is a join. The split and join can have different types.

If $FOLLOW(n_1) = \{ n_2 \}$ for $n_1, n_2 \in N$, then the pair $n_1, n_2$ demarcates a single entry single exit (SESE) region, where $n_1$ is the entry node and $n_2$ the exit node. However, set $FOLLOW(n_1)$ can be empty, as the following example illustrates.

**Example 2.** Consider the process flow graph in Figure 1. Table 1 shows the FOLLOW sets for the nodes in the process flow graph.

<table>
<thead>
<tr>
<th>Node n</th>
<th>FOLLOW(n)</th>
<th>Node n</th>
<th>FOLLOW(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>{Assess claim}</td>
<td>A3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Assess claim</td>
<td>{A1}</td>
<td>Send decision</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>A1</td>
<td>{A6}</td>
<td>Approve payment</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Monitor claim</td>
<td>$\emptyset$</td>
<td>Accept claim</td>
<td>{A5}</td>
</tr>
<tr>
<td>A4</td>
<td>{File report}</td>
<td>A5</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>File report</td>
<td>$\emptyset$</td>
<td>X2</td>
<td>Pay</td>
</tr>
<tr>
<td>X1</td>
<td>$\emptyset$</td>
<td>A3</td>
<td>Archive</td>
</tr>
<tr>
<td>Inspect vehicle</td>
<td>{A2}</td>
<td>Pay</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>A2</td>
<td>$\emptyset$</td>
<td>A6</td>
<td>{Archive}</td>
</tr>
<tr>
<td>X2</td>
<td>{A3}</td>
<td>end</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 3: Dominator tree for Figure 1

Table 1: FOLLOW sets for nodes in Figure 1
Table 2: Properties satisfied by acyclic process flow graphs

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>How satisfied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>For each pair of distinct nodes $n_1, n_2$, $FOLLOW(n_1) \cap FOLLOW(n_2) = \emptyset$</td>
<td>By default</td>
</tr>
<tr>
<td>P2</td>
<td>For each node $n$, $</td>
<td>FOLLOW(n)</td>
</tr>
<tr>
<td>P3</td>
<td>For each split $s$, there is no join $j$ such that $(s, j)$ is an edge in $E$</td>
<td>Refactoring transitive edges</td>
</tr>
<tr>
<td>P4</td>
<td>For each AND join $j$, there is a split $s$ such that $j \in FOLLOW(s)$</td>
<td>Removing cross-synchronization</td>
</tr>
</tbody>
</table>

1: procedure PREPROCESS($\mathcal{P}$)
2: $\mathcal{P} :=$ RemoveTransitiveEdges($\mathcal{P}$)
3: $\mathcal{P} :=$ InsertSplits($\mathcal{P}$)
4: $\mathcal{P} :=$ RemoveCrossSynchronization($\mathcal{P}$)
5: $\mathcal{P} :=$ InsertSplits($\mathcal{P}$)
6: end procedure

Figure 4: Preprocessing algorithm

Note that $FOLLOW$ sets were introduced by Baker [26] for program flow graphs. We adapted the definition slightly: the original definition does not require that for a split node $n$, each node in $FOLLOW(n)$ post-dominates $n$. Consequently, the original definition allows that a split node has more than one node in set $FOLLOW$. The definition allows the structuring algorithm described in Section 5 to duplicate nodes, which is not catered for by the original structuring algorithm of Baker.

Table 2 lists properties of $FOLLOW$ that are important for the algorithm described in Section 5. Properties P3 and P4 only hold after applying preprocessing steps defined in Section 4. In Appendix B we prove that the properties hold.

4. Preprocessing

We outline three preprocessing steps that have to be applied to an unstructured process model before invoking the structuring algorithm described in the next section (see Figure 4).

4.1. Refactoring transitive edges

Consider a split $s$ and join $j$. If there is an edge $(s, j)$ in $E$, and a path $p$ from $s$ to $j$ that does not include $(s, j)$, then edge $(s, j)$ represents a transitive relation, and we therefore call $(s, j)$ transitive. To simplify the definition of the algorithm in the next section, we refactor all transitive edges (Figure 5).

If $s$ is an AND split, then transitive edge $(s, j)$ is redundant and is therefore removed. If $s$ is an XOR split, then $(s, j)$ specifies a bypass edge: by taking $(s, j)$ join $j$ can be reached directly and the other nodes on path $p$ are passed by.
1: procedure RefactorTransitiveEdges($P$)  
2: for each transitive edge $(s, n) \in E$ do  
3: remove $(s, n)$ from $E$  
4: if $s$ is a XOR split then  
5: $d :=$ a dummy task such that $d \notin T$  
6: $T := T \cup \{d\}$  
7: $E := E \cup \{(s, d), (d, n)\}$  
8: end if  
9: end for  
10: return $P$  
11: end procedure  

Figure 5: Algorithm for removing transitive edges

A bypass edge is similar to an empty else-clause inside an if-then-else statement. For each bypass edge $(s, j)$, we insert a dummy task $d$ between $s$ and $j$, and replace edge $(s, j)$ with edges $(s, d)$ and $(d, j)$.

4.2. Inserting splits  
If a join $j$ does not occur in the FOLLOW set of any (split) node, sometimes a split $s$ can be inserted such that $j \in \text{FOLLOW}(s)$. Each join has, by definition, an immediate dominator $s$, so $s = \text{IDOM}(j)$, that is a split. An extra split $s'$ can be inserted immediately after $s$ such that $\text{FOLLOW}(s') = \{j\}$. Splits $s'$ gets the same connector type as $s$, so either $s, s' \in A$ or $s, s' \in X$.

**Example 3.** In Figure 6(a), AND join $A_2$ is not in the FOLLOW set of AND split $A_1$, but $A_1$ is the immediate dominator of $A_2$. AND split $A_4$ can be inserted immediately after $A_1$; see Figure 6(b). AND split $A_4$ corresponds to AND join $A_2$; so $\text{FOLLOW}(A_4) = \{A_2\}$. \hfill $\Box$

However, a split can only be inserted if there are two or more successor nodes of $s$ that are post-dominated by $j$. If $j$ post-dominates only one successor of $s$, then the inserted split would have only one successor and one predecessor $(s)$, and thus would be redundant. Note that if all successor nodes of $s$ are post-dominated by $j$, then $j \in \text{FOLLOW}(s)$; thus, no split needs to be inserted.

**Example 4.** In Figure 1 join $A_4$ is not in the FOLLOW set of $A_1$ but $A_1$ is the immediate dominator of $A_6$. However, only one successor of split $A_1$ is post-dominated by join $A_6$. Inserting a split that corresponds to join $A_3$ is not possible. \hfill $\Box$

The algorithm in Figure 7 details how splits are inserted. For each join $j$ that does not occur in the FOLLOW set of any node, a split $s'$ is inserted if $s = \text{IDOM}(j)$ and two or more successors of $s$ are post-dominated by $j$. The connector type of $s'$ (AND/XOR) is the same as the type of $s$. For every successor $n$ of $s$ that is post-dominated by $j$, $(s, n)$ is replaced with $(s', n)$ in $E$. Finally, edge $(s, s')$ is added to $E$. 
4.3. Removing cross-synchronization

After the second preprocessing step, a process flow graph may have an AND join that is not in any FOLLOW set, so for every node \( n \) we have \( j \notin \text{FOLLOW}(n) \). In that case, cross synchronization between parallel branches occurs at \( j \). For example, in Figure 1 AND join A4 is not in the FOLLOW set of any split. AND split A1 starts two parallel branches which are cross-synchronized by the path from A3 to A4.

Such cross synchronization cannot be expressed in a structured process [10, 11]. However, the cross synchronization can be expressed in a semi-structured process, which is a structured process with additional synchronization constraints between parallel blocks. While every correct process flow graph can be expressed in an equivalent semi-structured process [18], there is currently no translation that actually realizes this transformation.

We now define a two-step preprocessing rule that removes cross-synchronization edges, called \textit{links}, from the process model such that a semi-structured process can be constructed by the algorithm defined in the next section. To ensure that the behavior of the changed process flow is equivalent to the original process flow graph, we use additional synchronization conditions in the process flow graph to capture the constraints expressed by the removed links.

In the first step (Figure 8), the set \( \text{links} \subseteq E \) of cross-synchronization edges is computed. Initially, \( \text{links} = \emptyset \). To compute links, we first identify the AND split and AND join that ‘causes’ the link (see Figure 9). For an AND join \( j \) that is not in the FOLLOW set of any split, we first identify the split \( s' \) that is the immediate dominator of \( j \). That split starts a block that contains \( j \). The block is closed by the unique node \( j' \in \text{FOLLOW}(s') \).
1: **procedure** `InsertSplits(P)`
2:  
3:   for each join \( j \in A \cup X \) do
4:     if \( j \) is not in any \( FOLLOW \) set then
5:       \( s := IDOM(j) \)
6:       if two or more successors of \( s \) are post-dominated by \( j \) then
7:         \( s' := \) a new split such that \( s' \notin A \cup X \)
8:         if \( s \in A \) then
9:           \( A := A \cup \{s'\} \)
10:          else
11:            \( X := X \cup \{s'\} \)
12:          end if
13:        end if
14:      end for
doi
15:    for each successor \( n \) of \( s \) such that \( n \) is post-dominated by \( j \) do
16:      replace \((s, n)\) with \((s', n)\) in \( E \)
17:    end for
18:  end if
19: return \( P \)
20: end procedure

Figure 7: Algorithm for inserting splits

1: **procedure** `ComputeLinks(P)`
2:  
3:   \( links := \emptyset \)
4:   for \( j \in A \) that is an AND join do
5:     if there is no split \( s \in A \) such that \( j \in FOLLOW(s) \) then
6:       \( s' := IDOM(j) \)
7:       \( j' := IPDOM(s') \)
8:         for each \( s \in A \) that is an AND split do
9:           if \( FOLLOW(s) = \emptyset \) and \( j' = IPDOM(s) \) then
10:             for each predecessor \( p \) of \( j \) do
11:               if there is a path from \( s \) to \( p \) then
12:                 \( links := links \cup \{(p, j)\} \)
13:               end if
14:             end for
15:           end if
16:         end for
17:       end if
18:   end for
19: return \( links \)
20: end procedure

Figure 8: Algorithm for computing synchronization links
Since \( j \) is not in \( \text{FOLLOW}(s') \), there is a split \( s \) inside the block that allows to bypass \( j \) (see Figure 9). Since the process flow graph is correct, \( s \) has to be an AND node. (If \( s \) is a XOR node and during execution the second branch from \( s \) to \( j' \) not passing \( j \) is chosen, a deadlock occurs at \( j \).) However, \( j' \) is not an immediate post-dominator of \( s \) because of \( j \). Therefore \( \text{FOLLOW}(s) = \emptyset \). Node \( s \) is not unique, so there can be multiple splits that satisfy these constraints.

Every incoming edge \((x, j)\) of \( j \) such that the source \( x \) of the edge is on the path from \( s \) to \( j \) is a link (line 10 in Figure 8). The preprocessing algorithm in Figure 8 computes all links as specified above.

**Example 5.** AND join \( A4 \) in the process flow graph in Figure 1 does not occur in any FOLLOW set (see Table 1). The pattern in Figure 9 occurs as follows in Figure 1: \( j = A4, s = A3, s' = A1, j' = A6 \). Therefore \( \text{links} = \{ (A3, A4) \} \).

In the second step (see Figure 10), the process flow graph is changed. First, if the source of a link has a single predecessor \( x \) which is a task (line 4), then the source of the link can be replaced with \( x \). Symmetrically, if the target of a link has a single successor \( x \) which is a task, the target of the link can be replaced with \( x \) (line 9).

Next, the algorithm removes each synchronization link \((p, j)\) and adds \( \text{done}(p) \) to the synchronization condition of \( j \), expressed using function \( \text{synch} \). If \( j \) is target of multiple links, then the synchronization condition for \( j \) is a conjunction of \( \text{done} \) predicates, one for every link source (line 17). The empty synchronization condition is \( \text{true} \) (line 14).

If the source \( p \) of a synchronization edge \( e = (p, j) \) is an AND split, then \( e \) is removed from set \( E \) (line 19). Otherwise, \( p \) is a task or a join, so \( e \) is the only outgoing edge of \( x \). In that case removing \( e \) would destroy the path from \( p \) to \( \text{end} \) and violate the syntax of process flow graphs. Therefore, if \( p \) is not a split, \( e \) is replaced in \( E \) with edge \((p, j')\) (line 23). This edge does not add any new constraints: since \( \text{FOLLOW}(s) = \{j'\} \), the edge does not affect the control flow after \( j' \).

Since this change affects the definition of the process flow graph, the second preprocessing step is repeated, so some additional splits might be inserted (see Figure 4). The first preprocessing step does not need to be repeated.

**Example 6.** For the process flow graph in Figure 1, the set \( \text{links} \) returned by \text{ComputeLinks} is \( \{ (A3, A4) \} \). Since the single successor of \( A4 \) is task
procedure RemoveCrossSynchronization($P$)
  links := ComputeLinks($P$)
  for $(p, j) \in$ links such that $p \in A$ do
    if $p$ has a single predecessor $x \in T$ then
      replace $(p, j)$ with $(x, j)$ in links
    end if
  end for
  for $(p, j) \in$ links such that $j \in A$ do
    if $j$ has a single successor $x \in T$ then
      replace $(p, j)$ with $(p, x)$ in links
    end if
  end for
  for $n \in$ Nodes do
    sync($n$) := true
  end for
  for $(p, j) \in$ links do
    synch($j$) := synch($j$) $\land$ done($p$)
    if $p$ is a split then
      remove $(p, j)$ from $E$
    else// $p$ is a task or an AND join
      $s' = IDOM(j)$
      $j' = IPDOM(s')$
      replace $(p, j)$ with $(p, j')$ in $E$
    end if
  end for
  return $P$
end procedure

Figure 10: Algorithm for removing cross-synchronization

File report, algorithm RemoveCrossSynchronization replaces $(A3,A4)$ with $(A3,File report)$. Next, the algorithm computes synchronization condition

$synch(File report) = done(A3)$, which is used in Figure 2. All the other synchronization conditions equal true.

For the process flow graph in Figure 11, the edge leaving $D$ is the only synchronization link. Since the target of the synchronization link has a single successor task $F$, preprocessing algorithm RemoveCrossSynchronization replaces the synchronization link with $(D,F)$ and $sync(F) = done(D)$ (see Figure 12). □

After applying this preprocessing rule, each process flow graph satisfies property P4 (see Table 2) since the rule removes synchronization links from process flow graphs.

The next section describes how preprocessed process flow graphs are converted into structured processes.
5. Structuring algorithm

We define an algorithm that structures process flow graphs in an automated way. The input to the algorithm is a process flow graph that has been preprocessed as explained in Section 4. The output is a semi-structured process flow graph.

The algorithm is listed in Figure 13. It takes as input a process flow graph $P = (N, T, A, X, \text{start}, \text{end}, E, \text{synch})$, and the node $\text{current}$ that is to be processed. The algorithm returns a sequential block that starts with $\text{current}$. The initial call is $\text{ConstructStructuredProcess}(P, \text{start})$.

The algorithm has two main parts.

- In Part 1 (l. 3-22), the node $\text{current}$ is processed by creating a sequential block $P$ containing $\text{current}$ and all its indirect successors that are not in $\text{FOLLOW}(\text{current})$. By definition, these indirect successors are on the path from $\text{current}$ to the unique node in $\text{FOLLOW}(\text{current})$.

- In Part 2 (l. 24-33), the nodes in $\text{FOLLOW}(\text{current})$ are processed by recursively calling $\text{ConstructStructuredProcess}$; the resulting block is appended to $P$. If $\text{FOLLOW}(\text{current}) = \emptyset$ then under certain conditions (see l.29) the successor of $\text{current}$ is processed, in which case the process block starting with $\text{current}$ is duplicated in the structured process. Finally (l. 34), $P$ is returned as the structured composition for $\text{current}$.

We now explain these parts in more detail. We illustrate the different parts by Examples 7-11, which form a running example that refers to the process in Figure 1 after preprocessing.

Part 1. The algorithm first processes the node $\text{current}$, which creates a sequential block $P$ containing $\text{current}$ and all its (in)direct successors that are not in $\text{FOLLOW}(\text{current})$ There are three cases.
procedure ConstructStructuredProcess(P, current)
  // part 1
  if current is a split then
    children := ∅
    for each successor n of current do
      C_n := ConstructStructuredProcess(P, n)
      children := children ∪ {C_n}
    end for
    if current ∈ X then
      P := X(c_1, c_2, ..., c_n), where children = {c_1, c_2, ..., c_n}
    else if current ∈ A then
      P := A(c_1, c_2, ..., c_n), where children = {c_1, c_2, ..., c_n}
    end if
  else if current ∈ T then
    P := proc(current)
  else if current has one outgoing edge then
    if current ∈ A and exists n ∈ N s.t. done(current) is in synch(n) then
      P := proc(current)
    else
      P := skip
    end if
  end if
  // part 2
  if FOLLOW(current) ≠ ∅ then
    n := the unique node in FOLLOW(current)
    Q := ConstructStructuredProcess(P, n)
    P := P; Q
  else // FOLLOW(current) = ∅
    if current has as only successor n ∧ no FOLLOW set contains n
      Q := ConstructStructuredProcess(P, n)
      P := P; Q
    end if
  end if
  return P
end procedure

Figure 13: Algorithm for constructing structured processes
- If \( \text{current} \) is a split node (l. 3), then each successor \( n \) of \( \text{current} \) is processed in a for-loop (l. 5). By the first preprocessing step, \( n \) is not a join. Therefore, \( n \) is a task or a split or a loop node. By definition node \( n \) does not appear in the \( \text{FOLLOW} \) set of any node, so \( n \) is not processed in the second part of the algorithm. First, algorithm \( \text{CONSTRUCTSTRUCTURED-PROCESS} \) is recursively called for \( n \) and returns a block \( C_n \) that contains \( n \) (l. 6). Finally, block \( C_n \) is added to the set \( \text{children} \) (l. 7). After the for-loop has finished, all child blocks have been computed and a composite block \( P \) containing all child blocks in \( \text{children} \) is created. The type of \( P \) depends upon the connector type of \( \text{current} \) (l. 9 and l. 11).

Example 7. If \( \text{current} = A_1 \), then the algorithm creates two child blocks, one starting with \text{Inspect vehicle}, the other with \text{Accept claim}; see Figure 2.

- If \( \text{current} \) is a task (l. 14), then a process flow graph only containing \( \text{current} \) is created, denoted \( \text{proc}(\text{current}) \) (l. 15). Unlike the previous two cases, the successor nodes of \( \text{current} \) are now not processed immediately. Since \( \text{current} \) is not a split or a loop node, the unique successor node \( x \) of \( \text{current} \) is in some set \( \text{FOLLOW}(n) \), where \( n \) is a node. If \( \text{current} \) is the only predecessor of \( x \), then \( x \in \text{FOLLOW}(\text{current}) \), so \( \text{current} = n \). Otherwise, \( x \) is a join node and \( n \) is a split node. In both cases, node \( x \) is processed in l. 24-33 if node \( n \) is processed as \( \text{current} \).

Example 8. If \( \text{current} = \text{Inspect vehicle} \), then the algorithm creates a block only containing task \text{Inspect vehicle}. The unique successor of \text{Inspect vehicle} is \text{AND split A2} which is processed at l. 24; see Example 10.

- If \( \text{current} \) has one outgoing edge but is not a task (l. 16), then \( \text{current} \) is a join. If \( \text{current} \in A \) and \( \text{current} \) is referenced in a synchronization condition of a node \( n \in N \), i.e. \( \text{synch}(n) \) contains \( \text{done}(\text{current}) \), then block \( P \) has to contain a task \( \text{current} \). Therefore in line 18 a new dummy task \( \text{current} \) is created using \( \text{proc}(\text{current}) \). Otherwise, an empty block \( \text{skip} \) is created. The empty block is removed if it is concatenated with another block, so \( \text{skip}; Q \) is equivalent to \( Q \).

Example 9. If \( \text{current} = X_2 \), then the algorithm creates an empty block \( \text{skip} \). If \( \text{current} = A_3 \) then the algorithm creates at line 18 a block only containing dummy task \( A_3 \), since \( A_3 \) is an \text{AND split} and the synchronization condition of \text{File report} contains \( \text{done}(A_3) \) (cf. Example 6).

Part 2. Next, the algorithm processes set \( \text{FOLLOW}(\text{current}) \). There are two options.

- If \( \text{FOLLOW}(\text{current}) \neq \emptyset \) (l. 24), then by definition there is a unique node \( n \) such that \( \text{FOLLOW}(\text{current}) = \{n\} \). Variable \( n \) gets assigned this node \( n \). The algorithm is invoked for node \( n \) (l. 26), and the returned block is appended to the block \( P \) created for \( \text{current} \) (l. 27).
Example 10. If current = X2, then n = A3. The block P created for X2 is skip (cf. Example 9). The block Q created for A3 is A3;Send decision. The new block P at line 27 becomes A3;Send decision, since the empty block skip created for X2 is removed when concatenated with the block created for A3.

For current = Inspect vehicle, node n = A2. The block Q created for A2 is A(A3;Send decision, Approve payment;Pay). The next Example 11 explains how block Approve payment;Pay is constructed.

- If FOLLOW(current) = ∅ (l. 28), then if current has a single successor n for which there is no node x such that n ∈ FOLLOW(x), then n is processed next by invoking the algorithm for n (l. 30). Since current has only one successor n yet n ∈ FOLLOW(current), n has to be a join. By property P4, n ∈ X so n is a XOR join. The block created for n is appended to P.

Example 11. For current = Approve payment, set FOLLOW(Approve payment) = ∅ and its unique successor X3 is not contained in any FOLLOW set (cf. Table 1). The block Q created for X3 only contains Pay, therefore P becomes Approve payment;Pay.

Finally, block P is returned (l. 34).

Example 12. Applying the algorithm to the process flow graph in Figure 1 results in the semi-structured process flow graph visualized in Figure 2. Dummy task (A3) is needed to specify the cross-synchronization between the parallel branches using condition [done(A3)]. Example 9 explains how the algorithm generates dummy task A3.

The part of the process flow graph between X2, X3 and A6 is duplicated by the algorithm. Node X2 gets processed twice as a result of processing splits A2 and A5 (recursive invocation at l. 6). Node X2 is not processed at l. 30 when A2 (or A5) is current: although FOLLOW(A2) = ∅, node A2 (A5) has more than one successor.

Node X3 is also processed twice. The first processing of X3 starts at l. 6 when A5 is current. The second one starts if Approve payment is processed as current, explained in Example 11. Since X3 is processed twice, two duplicate blocks containing Pay have been created for the semi-structured process in Figure 2. Without introducing duplicates, no structured process equivalent exists for the process flow graph, unless auxiliary variables are used [11].

We prove the correctness of the algorithm in two propositions, the proofs of which are in Appendix B. First we show that every node in an input process flow graph gets processed by the algorithm.
18

Proposition 1. Let $P$ be a process flow graph that is an input parameter to the initial invocation of algorithm ConstructStructuredProcess. Each node $n$ of $P$ is processed at least once by ConstructStructuredProcess, i.e., there is an invocation with current $= n$.

The algorithm can process a node multiple times, subject to a maximum bounded of the number of its incoming edges. For instance, $X2$ in Figure 1 is processed two times because it has two incoming edges, from $A1$ and $A2$. Each processing of a node $n$ results in a separate block in the structured process (cf. line 6). The algorithm therefore runs in time linear to the number of nodes and edges of the input process flow graph. However, the worst-case time complexity of the whole approach is slightly higher, since computing dominators takes almost linear time [27], but still polynomial.

Next, the main proposition shows that the algorithm converts a process flow graph satisfying properties P1-P4 into an equivalent structured processes, neither reducing nor adding behavior. Note that properties P1-P4 hold for every preprocessed process flow graph.

Proposition 2. If a process flow graph $P$ satisfies properties P1-P4 and is correct, then $P$ is equivalent to the structured process returned by ConstructStructuredProcess for $P$.

Next, we discuss the validation of the approach.

6. Evaluation

To evaluate the feasibility of the method, we have implemented the preprocessing rules and the algorithm, including its extensions in Appendix A, in StructXPDL, a Java-based tool which reads process models in XPDL format [28], an XML standard for exchanging process models. To compute dominators, we use the polynomial time algorithm developed by Lengauer and Tarjan [27].

To evaluate the utility of the method, we compare it with the approach developed by Polyvyanyy et al. [18]. We perform the comparison by applying the method, using the tool StructXPDL, to eleven correct benchmark examples developed for the BPStruct tool [29] — which implements the approach developed by Polyvyanyy et al. [18] — and comparing the outcomes. The models are acyclic and contain mixtures of AND and XOR splits and joins. A companion online appendix lists the examples as well as the outputs generated by the tool. Table 3 compares the performance of BPStruct and StructXPDL on these benchmark examples. Several interesting conclusions can be drawn from the comparison.

First, for the seven models that BPStruct can structure, StructXPDL also outputs a structured process model. Moreover, the structured process models that are constructed by BPStruct and StructXPDL for each of these seven models are identical, which indicates the StructXPDL method is sound for these examples. In addition, StructXPDL outputs semi-structured process models for
Table 3: Comparing BPStruct and StructXPDL on benchmark examples

<table>
<thead>
<tr>
<th>Input model</th>
<th>BPStruct output</th>
<th>StructXPDL output</th>
<th>identical?</th>
<th>identical after extra preprocessing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7817</td>
<td>structured</td>
<td>structured</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>7818</td>
<td>structured</td>
<td>structured</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>7819</td>
<td>–</td>
<td>semi-structured</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>7820</td>
<td>structured</td>
<td>semi-structured</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>7821</td>
<td>–</td>
<td>semi-structured</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>7822</td>
<td>structured</td>
<td>structured</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>7823</td>
<td>–</td>
<td>semi-structured</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>7824</td>
<td>structured</td>
<td>structured</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>7825*</td>
<td>structured</td>
<td>structured</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>7826</td>
<td>–</td>
<td>semi-structured</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>7827</td>
<td>structured</td>
<td>structured</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

*Loop modified into repeat-until structure

The four models that BPStruct cannot structure. All these four structured models contain cross-synchronization and are constructed using the algorithm detailed in Section 5, which clearly shows the benefit of the method. For instance, Figure 14(a) shows the original model 7826, which does not have a structured equivalent, so BPStruct does not generate an output model. However, our algorithm constructs after preprocessing the semi-structured process shown in Figure 14(c). Notice that for a reader Figure 14(c) is much more understandable than Figure 14(a). In the figure, X is the label of the XOR node that is the source of the synchronization edge in Figure 14(b). Node X is included by processing line 18 in Figure 13. Semantically, done(X) is true when node X has been activated by any of its incoming edges. The process can be simplified further by replacing the parallel subprocess containing X and skip with the process containing a single task X.

Second, for some models StructXPDL outputs a structured process that is different from the structured process created by BPStruct. In all these cases, the input model contains a subgraph consisting of j splits and n joins for n, j ≥ 2, where for every split-join pair, there is an edge from the split to the join, and every edge entering the subgraph enters a split while every edge leaving the subgraph exits a join. To illustrate this subgraph structure, Figure 15(a) shows model 7827, where j = n = 2. Since the splits have type AND and the joins are XOR, the algorithm duplicates c and d; it returns the structured process shown in Figure 15(b).

To remove duplication, we experimented with an additional preprocessing rule, which replaces the subgraph with one join and subsequent split; Figure 15(c) shows the process flow graph of (a) after additional preprocessing. If StructXPDL with the additional preprocessing rule is applied to model 7827, it produces the structured process shown in Figure 15(c), so the output is then
Figure 14: Preprocessing of benchmark model 7826
identical to the output model produced by BPStruct. More generally, with this additional preprocessing rule StructXPDL and BPStruct produce identical models for all input models listed in Table 3.

We also applied the tool to another unstructured process which contains several cross-synchronizations between parallel branches [30]. While the model cannot be structured with BPStruct, StructXPDL outputs a semi-structured process model.

From the comparison based on these examples, we conclude that the proposed method outputs (semi-)structured process models for all (correct) input models. The BPStruct tool is able to convert only some of the process models to equivalent structured models, because it restricts itself to structured models. Since cross-synchronization between parallel branches occurs quite frequently, this is a useful improvement over the current state of the art implemented in the BPStruct tool. Next, the comparison shows that the algorithm does not always return minimal models. But for all these cases, if an additional preprocessing rule is used, the algorithm does return a minimal model that is equivalent to the models created by BPStruct.

7. Related Work

Early works addressing structured process models are [10, 11], in which the class of structured process models is defined and their expressiveness is analyzed. Informal transformations are proposed to convert unstructured process patterns to structured patterns, extending existing translation patterns used to structure sequential programs. However, no complete overall conversion approach is proposed in these early works.

Ouyang et al. [17] focus on recognizing structured patterns in unstructured process models and mapping them to structured BPEL fragments, but they do not address how an unstructured process model can be structured. For example, the unstructured process model in Figure 1 cannot be structured using the approach of Ouyang et al.

Other related work [31] proposes to decompose a process flow graph into SESE regions, based on transformation rules that build the regions incrementally. A SESE region has a unique entry and a unique exit node. Our approach does not use SESE regions; instead, the complete process flow graph is processed. Citation [31] also sketches how each atomic SESE region (not containing any other SESE region) can be converted into a structured process but does not provide a complete specification for that conversion. In particular, it is not discussed in what circumstances duplications are required. The algorithm in Section 5 defines precisely when duplications of tasks and blocks are made.

The work most closely related to ours is by Polyvyanyy et al. [18], who outline a method for structuring acyclic process models; the method has been implemented in the BPStruct tool [29], discussed in Section 6. In the BPStruct method, first a process model is translated into Petri nets and decomposed into a refined process structure tree (RPST) consisting of SESE regions. Next, each
Figure 15: Additional preprocessing of benchmark model 7827.
Table 4: A comparison of our approach with the BPStruct approach [18]

<table>
<thead>
<tr>
<th>Aspect</th>
<th>BPStruct approach</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Petri nets</td>
<td>Process flow graphs</td>
</tr>
<tr>
<td>Basis for structuring</td>
<td>Petri-net unfoldings, ordering relations graphs</td>
<td>Dominators</td>
</tr>
<tr>
<td>Output</td>
<td>Structured process models</td>
<td>Semi-structured, structured process models</td>
</tr>
<tr>
<td>Duplicate tasks</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>Optional tasks</td>
<td>Not allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>Loops</td>
<td>Not allowed</td>
<td>SESE loops allowed</td>
</tr>
<tr>
<td>Time complexity</td>
<td>Exponential</td>
<td>Polynomial</td>
</tr>
</tbody>
</table>

A region is unfolded into an equivalent region that contains no XOR joins. The unfolding may lead to duplication of tasks. A graph containing ordering relations between tasks is derived from the unfolding. The ordering relations graph can be translated (sometimes) into a structured region. The complete approach combines techniques from different areas, in particular Petri net unfoldings and ordering relations graphs. In a follow-up paper, Polyvyanyy et al. [32] refine the approach to obtain maximally structured process models, i.e., models that contain as many structured subprocesses as possible.

There are several differences between BPStruct and our work. The worst-case time complexity of BPStruct is exponential, though the authors argue that the average complexity is much lower in practice. In contrast, our approach runs in polynomial time. Another difference is that they do not consider semi-structured process models which can be generated with the algorithm proposed in this paper. Finally, for process models that contain optional tasks the ordering graph does not capture the process behavior accurately since the ordering graph will treat all tasks as mandatory. For instance, a process in which first task A is done, then either B or nothing, and finally C, translates using the ordering graph into a structured process in which A, B, C are done in sequence. Like the extended BPStruct approach [32], our approach constructs maximally structured process models. A summary of the comparison is shown in Table 4.

Finally, there is a large body of work on structuring control flow graphs (see [33] for an overview), but these are sequential while the process flow graphs underlying process models contain parallelism. There are a few papers that analyze concurrent flow graphs [34, 35]. However, these flow graphs are derived from programs that contain structured parallelism, i.e. a block containing parallelism is demarcated with a `cobegin` and a `coend` statement, and different blocks containing parallelism are either disjoint or properly nested. In contrast, the process flow graphs in this paper can contain arbitrary, unstructured parallelism, for instance the example presented in Figure 1.

The contribution of this paper is the formal definition of a polynomial time algorithm for converting correct, unstructured process models into equivalent (semi-)structured ones.
8. Conclusion

We have presented an algorithm that converts any correct, unstructured process flow graph into a (semi-)structured process. Such processes are easier to understand and implement than unstructured ones. Process flow graphs extend control flow graphs with unstructured but bounded concurrency and underly process model notations such as Petri nets [36], UML activity diagrams [16] and BPMN [15]. Semi-structured process models are structured processes that contain concurrency and cross-synchronization between parallel branches and underly notations such as BPEL [3] and OWL-S [4]. The algorithm enables a smooth translation from user requirements on business processes documented in process flow graphs into deployable process models that support the business processes. One drawback is that the algorithm sometimes generates duplicate parts when a structured model without duplicates may also exist, so the models are not always minimal. For the examples we checked, the duplicates can be removed using one additional preprocessing rule.

Further work includes adding more pre- and post-processing rules to our prototype. It would also help to extend the translation to other operators than sequence, choice, and parallelism. For instance, BPMN uses the notion of an OR split: a connector in which the number of outgoing edges taken is determined at run-time when the connector is visited. Implementing this approach in an existing workflow system is also an interesting exercise.

References


Appendix A. Loops

As explained in Section 1, the main text considers acyclic process flow graphs to simplify the exposition. This Appendix shows how to extend the algorithm of Section 5 to deal with process flow graphs with structured loops, in which each loop has a single entry and a single exit point. Other papers [36, 20, 21] already discuss techniques to turn a process model with unstructured loops into an equivalent process model in which each loop has a single entry and a single exit point. These techniques are complementary to our approach and can be easily integrated.

Appendix A.1. Definitions

We introduce additional definitions to deal with loops. A back edge is an edge \((x, y) \in E\) in the process flow graph such that \(y\) dominates \(x\) [19], thus every path from \(\text{start}\) to \(x\) passes through \(y\). Intuitively, a back edge \((x, y)\) closes a cycle that is started at \(y\). An edge that is not a back edge is called a forward edge.

Example 13. Figure 1 only has forward edges. Figure A.16 has one back edge: \((X3,X2)\). Note that node \(X2\) dominates node \(X3\).

A node \(l\) that is the target of some back edge is called a loop node. The modified algorithm defined below creates a repeat-until statement for \(l\). In a correct process flow graph, each loop node is a XOR node that has multiple
incoming edges. To simplify the presentation, we require that each loop node has a single outgoing forward edge, so each loop node is a XOR merge. If a node violates this constraint because the loop node has multiple outgoing forward edges, the graph can be easily repaired by splitting the loop node in a XOR merge and subsequent XOR split.

For a non-loop node \( n \), its loop head, denoted \( \text{HEAD}(n) \), is the most nested loop node \( l \) such that \( l \) strictly dominates \( n \) (so there is a path from \( l \) to \( n \)) and there is a reverse path from \( n \) to \( l \) via a back edge \((x, l)\), where \( x \in N \). In the constructed semi-structured process, \( n \) is contained in the body of the repeat-until statement constructed for \( \text{HEAD}(n) \). If \( n \) is not contained in a loop, \( \text{HEAD}(n) \) is undefined. For convenience, if \( \text{HEAD}(n_1) \) and \( \text{HEAD}(n_2) \) are undefined, then \( \text{HEAD}(n_1) = \text{HEAD}(n_2) \) by definition. Note that for a loop node \( l \), \( \text{HEAD}(l) \) is either another loop node or undefined.

**Example 14.** For Figure A.16, \( X_2 \) is header for nodes Repair damage, Check repair, and \( X_3 \). All the other nodes, including \( X_2 \) itself, have no header. In particular, \( X_2 \) is not header for Determine bill since there is no path from Determine bill to \( X_2 \) via \( X_3 \).

For a loop node \( l \), so some back edge enters \( l \), define

\[
\text{FOLLOW}(l) = \{ n | \text{HEAD}(n) = \text{HEAD}(l) \land \text{HEAD},IDOM(n) = l \}.
\]

So \( n \in \text{FOLLOW}(l) \) if and only if \( n \) and \( l \) have the same loop header and the immediate dominator of \( n \) is in a loop headed by \( l \). In that case, we call node \( \text{IDOM}(n) \) a loop-exit for the loop started at \( l \). By definition, \( \text{IDOM}(n) \) is a split node. We require that \( \text{IDOM}(n) \) is a XOR node; if \( \text{IDOM}(n) \) were an AND node, each execution would remain in the loop after reaching \( \text{IDOM}(n) \) and in parallel start a new copy of the subprocess started at \( l \). This results in unbounded many instantiations of the subprocess, which cannot be expressed in structured process models.

Next, we have to redefine the existing definition of \( \text{FOLLOW} \). For every other node \( n_1 \in N \), so \( n_1 \) is not a loop node, define

\[
\text{FOLLOW}(n_1) = \{ n_2 | n_1 = \text{IDOM}(n_2) \land n_2 = \text{IPDOM}(n_1) \land \text{HEAD}(n_1) = \text{HEAD}(n_2) \}.
\]

So for a non-loop node \( n \), each node \( y \) in the \( \text{FOLLOW} \) set of \( n \) must have the same loop header as \( n \).

**Example 15.** For Figure A.16, for example \( \text{FOLLOW}(X_2) = \{ \text{Determine bill} \} \), \( \text{FOLLOW}(\text{Repair damage}) = \{ \text{Check repair} \} \), \( \text{FOLLOW}(\text{Check repair}) = \{ X_3 \} \), and \( \text{FOLLOW}(X_3) = \emptyset \). Furthermore, \( X_3 \) is a loop-exit for the loop started at \( X_2 \).

In the sequel we only consider process flow graphs in which each loop node \( l \) has exactly one node in its \( \text{FOLLOW} \) set. This means that in addition to the properties listed in Table 2, we have new property:
For each loop node $l$, $|FOLLOW(l)| = 1$.

Furthermore, to simplify the presentation, we only consider process flow graphs with repeat-until loops, which have a single exit point. That is, for a loop node $l$ there is exactly one exit point $x$ and $x$ is a XOR split that has an outgoing backedge to $l$, so $(x, l) \in E$. For structuring loops with multiple exits, the approaches outlined in [20, 21] can be used.

To encode loops in structured processes, we extend the notation from Section 3 with construct `repeat P until grd`, which specifies that block $P$ is executed until condition $grd$ holds.

Appendix A.2. Modification of Algorithm for handling loops

We add the following else-if clause after line 13 in Figure 13:

```
else if current is a loop node with FOLLOW(current) = \{n\} then
  s := the unique successor of current
  $P_x := ConstructStructuredProcess(P, x)$
  $x :=$ the loop-exit of current
  $P := repeat P_x$ until condition on edge $(x, n)$
end if
```

If `current` is a loop node (l. 1), then by P5 there is a unique node $s$ that is the successor of `current`. For $s$, a block $P_s$ is created by invoking `ConstructStructuredProcess(P, s)` (l. 3). By assumption the loop has a single exit point $x$, which is a split. There is an edge from $x$ to a unique node $n$ outside the loop such that $FOLLOW(current) = \{n\}$. If node $x$ is reached and the condition annotated on $(x, n)$ is true, the loop is exited. Since we focus on repeat-until loops, split $x$ has no outgoing forward edges that target nodes inside the loop. Therefore, $x$ closes the block started at $s$ and the block `repeat P_s` until condition on $(x, n)$ represents the entire loop.

Appendix B. Proofs of Section 5

This Appendix contains the proofs of the propositions defined in Section 5. We first prove new Proposition 3, used in the other proofs, that states that properties P1-P5 (cf. Table 2) hold for preprocessed process flow graphs.

**Proposition 3.** After preprocessing each process flow graph $P = (N, T, A, X, \text{start}, \text{end}, E, \text{synch})$ satisfies properties P1-P5.

**Proof.** The first two properties hold for any process flow graph. Property P3 holds after refactoring transitive edges. Property P4 holds after synchronization links have been removed from the set $E$ of edges in a process flow graph. Property P5 follows from our definition of loops.

We now prove the propositions stated in Section 5.

**Proof of Proposition 1.** Let $n$ be an arbitrary node of $P$ that is processed by the algorithm `ConstructStructuredProcess` as `current`. We show that each successor node of `current` is processed too. Since the process flow graph is
connected, and in the initial invocation the start node is processed as current, this proves the claim.

There are two cases:

1. If current is a split, then each successor n is processed directly (l. 6).
2. If current is a task or an AND/XOR node with one outgoing edge, then current has by constraint a unique successor x. In that case, by P2 either
   (i) \text{FOLLOW}(current) = \{x\} or (ii) \text{FOLLOW}(current) = \varnothing. (i) By P1, there is no other node n' such that x \in \text{FOLLOW}(n'). Node x is therefore processed exactly once at l. 26. (ii) There are two subcases for x. (a) Either there is a split s such that \text{FOLLOW}(s) = \{x\}. Then x is processed when s is processed as current. (b) Or x is not in the \text{FOLLOW} set of any node. Then by P4, x is not in A, therefore x \in X. Since x is the only successor of current, node x gets processed at l. 30. □

**Proof of Proposition 2.** (Sketch.) By preprocessing link edges are removed from \(P\). It is straightforward to check that the synchronization conditions created by Algorithm \text{RemoveCrossSynchronization} preserve the synchronization constraints expressed by the removed link edges. We now focus on the correctness of algorithm \text{ConstructStructuredProcess}.

We prove the proposition for any process flow graph \(P\) obtained after preprocessing. We consider the different blocks (parts) of the semi-structured process flow graph created by \text{ConstructStructuredProcess} for \(P\). Each block has a single entry and a single exit node such that each edge from a node outside the block that targets a node inside the block must target the entry node, and each edge from a node inside the block to an outside node must leave the exit node.

Let \(P = (N, T, A, X, start, end, E, synch)\) be a process flow graph after preprocessing and let \(n \in N\) be a node of \(P\). The block created by \text{ConstructStructuredProcess} when processing \(n\) as current corresponds to a subgraph of \(P\). We first identify this subgraph, and then show that each subgraph is equivalent to its corresponding block.

First, observe that if \text{ConstructStructuredProcess}(P, n) is invoked, then processing \(n\) can result in a recursive invocation of \text{ConstructStructuredProcess} in which an (indirect) successor of \(n\) is processed as current and also included in the block for \(n\). For a node \(n \in N\), we now define a set \text{range}(n) containing all nodes processed as current by the algorithm when the initial invocation is \text{ConstructStructuredProcess}(P, n). Formally, \text{range}(n) is the smallest non-empty set of nodes \(S \subseteq N\) satisfying

- \(n \in S\);
- if \(x\) is a forward successor of some node in \(S\), and either:
  - \(x\) is not in the \text{FOLLOW} set of any node in \(N\), or
  - \(x\) is in the \text{FOLLOW} set of a node \(y \in S\),
  then \(x \in S\).
The construction of $range(n)$ stops when one or more nodes $y$ are reached such that $y \in FOLLOW(z)$ for some node $z \notin range(n)$. Such a node $y$ is processed when $\text{ConstructStructuredProcess}(\mathcal{P}, z)$ is invoked, and is therefore not contained in the block created for $n$.

For each node $n \in N$, we define the subgraph $\mathcal{S}$ that is induced by $range(n)$. Formally, the subgraph $\mathcal{S}$ of $\mathcal{P}$ induced by $range(n)$ is a tuple $(T', A', X', start', E', \text{synch}')$, where:

- $T' = T \cap range(n)$ is a set of tasks,
- $A' = A \cap range(n)$ is a set of AND nodes,
- $X' = X \cap range(n)$ is a set of XOR nodes,
- $start' = n$ is the unique start node,
- $E' = E \cap (range(n) \times range(n))$ is the control flow (transition) relation,
- $\text{synch}' = \text{synch} \cap (A \rightarrow BExp)$.

The subgraph $\mathcal{S}$ typically has no unique end node. For instance, for $A_1$ and $A_2$ in Figure 1 after preprocessing, so edge $(A_3,A_4)$ has been removed, we have $range(A_2) = range(A_5) = \{X_2,A_3,\text{Send decision},\text{Approve payment},X_3,\text{Pay}\}$, so the range contains end nodes $\text{Send decision}$ and $\text{Pay}$.

To prove the proposition, we now prove the following claim: for each node $n \in N$, the subgraph $\mathcal{S}$ induced by $n$ is equivalent to the structured process returned by $\text{ConstructStructuredProcess}(\mathcal{P}, n)$. The proof is by induction on the nesting level of the induced subgraphs. A subgraph induced by node $n$ is nested inside a subgraph induced by node $n'$ if $range(n) \subset range(n')$. The base case, where the subgraph induced by $n$ does not contain any other subgraph, i.e. $range(n) = \{n\}$, is trivially true.

For the induction case: we prove the claim for a subgraph induced by node $n$, assuming the claim holds for all subgraphs nested inside the subgraph induced by $n$. We consider the different cases for $n$.

1. $n$ is an AND or XOR split. If the algorithm $\text{ConstructStructuredProcess}$ is invoked when $n$ is current, an AND or XOR block is created (l. 3 and further), depending on the connector type of $n$. The block created for each successor $x$ of $n$ is a child of this block. By properties P3 and P4, $x$ is not an AND join, so the block for $x$ does not need synchronization with another block before it can start. By the induction hypothesis, the block created for $x$ at l. 5 is equivalent to the subgraph induced by $x$.

The block opened for $n$ can be closed in two different ways. If $FOLLOW(n) = \{y\}$ then $y$ is by definition a join that closes the block $P$ started at $n$ and $y \in range(n)$. By P3, $y$ is not an immediate successor of $n$, so no block for $y$ has been created at line 5. Therefore for $y$ only one block $Q$ is created, at line 26. By the induction hypothesis, $Q$ is equivalent to the subgraph induced by $y$. Consequently, $P;Q$ is equivalent to the subgraph induced by $n$.

If $FOLLOW(n) = \emptyset$ then the block created for $n$ is $P$. In particular, $P$ does not contain the post dominator $\text{IPDOM}(n)$ of $n$. By construction, set
range(n) does not include IPDOM(n). Consequently, P is equivalent to the subgraph induced by n.

2. n is a loop node. (Defined in Appendix A.) By P5, n has a unique successor s. Next, there is a unique exit point x and FOLLOW(n) = {e} with (x, e) ∈ E. By the induction hypothesis, the subgraph started at s is equivalent to the block Ps returned by recursively calling the algorithm CONSTRUCT-STRUCTUREDPROCESS at l. 3. Since the loop has by constraint only one exit point, node x has no outgoing forward edges that target nodes in the loop headed by n. The block Ps is therefore closed by x and exited when the condition on edge (x, e) is true. Therefore block \( P = \text{repeat } P_s \text{ until condition on edge } (x, n) \) (l. 5) is equivalent to the first part of subgraph induced by n, which ends at e. Next, a block Pe is created by recursively invoking the algorithm CONSTRUCT-STRUCTUREDPROCESS at l. 26 and Pe is appended to P. By the induction hypothesis, block Pe is equivalent to the subgraph induced by e. The resulting block P is therefore equivalent to the entire subgraph induced by n.

3. n is a task, or an AND or XOR node with one outgoing edge. If the algorithm CONSTRUCT-STRUCTUREDPROCESS is invoked when n is current, a block n is created if n is a task (l. 15) and a block skip is created if n is an AND/XOR node with one outgoing edge (l. 20).

Next, let y be the single successor of n in \( \mathcal{P} \). There are two subcases.

3.1 FOLLOW(n) = \{y\}. By the induction hypothesis, for the subgraph induced by y an equivalent block Q is created, by recursively calling the algorithm CONSTRUCT-STRUCTUREDPROCESS (l. 26). The block n;Q or skip;Q is therefore equivalent to the subgraph induced by n, depending on whether n is a task or an AND/XOR node.

3.2 FOLLOW(n) = \ø. Since y \not\in FOLLOW(n), y is a join node. There are two further subcases.

3.2.1 No FOLLOW set contains y. By the induction hypothesis, for the subgraph induced by y an equivalent block Q is created, by recursively calling the algorithm CONSTRUCT-STRUCTUREDPROCESS (l. 30). By property P4 y is not an AND join, so y is a XOR join. As the process flow graph is correct, there is no lack of synchronization at y, and P;Q is equivalent to the subgraph induced by n.

3.2.2 y \in FOLLOW(z) for some node z \neq n. In that case, the condition of the if-statement at l. 29 is false, and the complete block P created for n is either P = n, if n is task, or P = skip if n is an AND/XOR join. By construction y \not\in range(n), so P is the block equivalent to the subgraph induced by n.

Thus, by induction and case by case enumeration, we have shown that the proposition is true. \(\square\)