On Separation of Concurrency and Conflicts in Acyclic Process Models

Felix Elliger, Artem Polyvyanyy, and Mathias Weske
Hasso Plattner Institute at the University of Potsdam, Germany
Felix.Elliger@student.hpi.uni-potsdam.de
(Artem.Polyvyanyy,Mathias.Weske)@hpi.uni-potsdam.de

Abstract: Recently, a new approach for structuring acyclic process models has been introduced. The algorithm is based on a transformation between the Refined Process Structure Tree (RPST) of a control flow graph and the Modular Decomposition Tree (MDT) of ordering relations. In this paper, an extension of the algorithm is presented that allows to partially structure process models in the case when a process model cannot be structured completely. We distinguish four different types of unstructuredness of process models and show that only two are possible in practice. For one of these two types of unstructuredness an algorithm is proposed that returns the maximally structured representation of a process model.

1 Introduction

The Business Process Modeling Notation (BPMN) offers a lot of concepts to model a business process. In general, for decision points and the introduction of several concurrent tasks the use of gateway nodes is suggested. A gateway in BPMN is either a *split* or a *join*. In the former case it has multiple outgoing edges, whereas in the latter it has multiple incoming edges. Each gateway has a type that defines a respective split or join behavior. However, abusive or unexperienced usage of gateway constructs may lead to complex, and even incorrect models. For a better and more intuitive understanding of process models it is preferable to have them obey to certain structures. Therefore, [KtHB00] defines the notion of *(well)-structured workflows*. In such workflows every split node has a corresponding join node of the same type, such that they form a single-entry-single-exit (SESE) region, also called a *block*. If a process model is not *well-structured*, we call it *unstructured*.

As outlined in [PGD10], transforming an unstructured process model into a behaviorally equivalent structured one is appealing from various perspectives. In addition to the motivation in [PGD10], Mendling et al. [MRvdA10] assess structured modeling as the most important guideline for process models. A pleasant side effect, concerning the understandability of a process model, is the reduction of routing paths, which is another guideline shown in [MRvdA10].

Figure 1 shows an example transformation. Figure 1a depicts the original process

Postprint

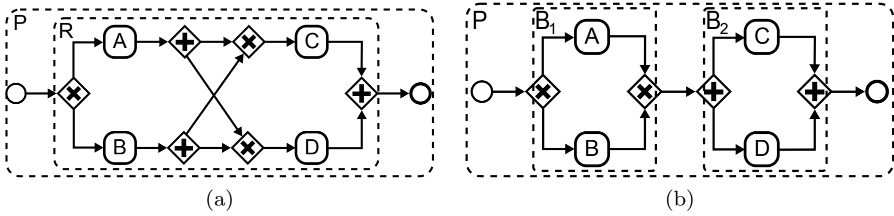

Figure 1: (a) An unstructured process model, and (b) its well-structured representation

model. Obviously, this model is unstructured. Firstly, the initial split and the final join have different types. Furthermore, the branches have connections between them which does not fit the defintion of *well-structuredness.* Figure 1b shows a well-structured process model whose behavior is *equivalent* to the behavior of the process model in Figure 1a, where *equivalent* implies that all concurrent activities of one process are also concurrent in the other one.

The algorithm for structuring process models, which is presented in [PGD10], has a set of limitations. The limitations include the restriction to a subset of BPMN elements that can be used and the absence of loops. Furthermore, the algorithm does not return a result if not the whole process model can be well-structured. This absence of a result is probably a crucial limitation, since there are process models that cannot be completely well-structured, but still contain some parts which can be replaced by an equivalent well-structured representation. An example for this type of process models is shown in Figure 2.

In this paper, an extension of the algorithm in [PGD10] is described that is capable of creating a valid result, even in the case of such inherently unstructured models. This result is the *maximally structured representation*, i.e., all parts of the model that can be structured, are represented as blocks. However, the extension does not cover all unstructured process models. We distinguish between four types of unstructured process models and show that only two of them have to be considered in practice. For one of these two an algorithm is presented that allows for the maximally structured representation of the process model.

The remainder of the paper is structured as follows. Section 2 introduces the algorithm of [PGD10] and describes the required preliminaries. In Section 3, we classify unstructured process models and describe the extension of the algorithm. The paper closes with conclusion.

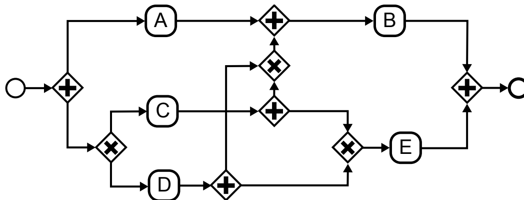

Figure 2: An inherently unstructured process model

2 Preliminaries

This section presents preliminaries that are required for the understanding of the algorithm in [PGD10] and its extension: First, Section 2.1 formally defines the notion of a process model and presents the Refined Process Structure Tree—a technique for parsing a process model into a hierarchy of SESE regions. Afterwards, Section 2.2 discusses Petri nets and their unfoldings. Finally, Section 2.3 discusses the notion of ordering relations and presents the Modular Decomposition Tree.

2.1 Process Models and the Refined Process Structure Tree

As already mentioned, the algorithm in [PGD10] imposes certain restrictions on the process model structure and the allowed modeling constructs. These restrictions are not lifted in this paper, i.e., we only consider acyclic sound process models that solely consist of *activities*, *AND-gateways*, and *XOR-gateways*. Following these restrictions, we define a process model as follows.

Definition 2.1 (Process model)
A *process model* is a tuple $W = (A, G^+, G^\times, C, \mathcal{A}, \mu)$, where A is a non-empty set of *activities*, G^+ is a set of *AND-gateways*, $G^\times$ is a set of *XOR-gateways*. $G = G^+ \cup G^\times$ is the set of all gateways of the process model. $Z = A \cup G$ is the set of all nodes of the process model. $C \subseteq Z \times Z$ defines the control flow relation. $\mathcal{A}$ is a set of names and $\mu : A \rightarrow \mathcal{A}$ is a function that assigns names to the activities.

Process models, as representatives of directed graphs, can be decomposed into SESE regions. In [VVK08, VVK09, PVV10], the *Refined Process Structure Tree* (RPST) is proposed as a unique hierarchical decomposition of a process model into SESE regions, where a region is defined by a subset of the control flow relation. The important fact besides the uniqueness of the decomposition is that identified regions do not overlap, i.e., given two SESE regions derived by the RPST either one region is completely contained in another region or these regions are disjoint.

According to its inner structure, each SESE region has a type. A *trivial (T)* region consists of a single control flow edge. A *polygon (P)* region refers to a sequence of regions. A *bond (B)* is a set of regions that share two common nodes, one as entry and one as exit. Typical examples of bonds are well-structured blocks in a process model with a split and a corresponding join gateways as common nodes. All other regions are referred to as *rigids (R)*. Rigid regions represent the unstructured parts of a process model. Therefore, if we talk about unstructured process models, or unstructured fragments of models, we talk about rigid regions of a process model.

In Figure 1a and Figure 1b, SESE regions are visualized using dashed rectangles with rounded corners. The letters in the top left corner indicate the types of the regions, e.g., P is a polygon, B denotes a bond, and R stands for a rigid. Note that trivial regions and polygon regions composed of two trivial regions are not shown in the figures.

2.2 Petri Nets and Unfoldings

The execution semantics of process models is defined by mapping process models to formal process languages. In [PGD10] and for our purposes, we map process models to Petri nets. Such a translation is described in detail by Dijkman et al. in [DDO07] or can be sufficiently looked up in [PGD10].

Definition 2.2 (Petri net)
A *Petri net*, or a *net*, is a tuple $N = (P, T, F)$ with P and T as finite disjoint sets of places and transitions, and $F \subseteq (P \times T) \cup (T \times P)$ as the flow relation.

For a node $n \in P \cup T$, we define the set of inputs $\bullet n = \{x \mid (x, n) \in F\}$ and the set of outputs $n\bullet = \{x \mid (n, x) \in F\}$. If n has multiple inputs $(|\bullet n| > 1)$, we call n a *join*. If n has multiple outputs $(|n\bullet| > 1)$, we call n a *split*. If n is a place (transition) we say that n is an *XOR- (AND-) split/join*. With F^+ we denote the transitive closure of the flow relation.

Definition 2.3 (Labeled net)
A *labeled net* is a tuple $N = (P, T, F, \mathcal{T}, \lambda)$, where (P, T, F) is a Petri net, $\mathcal{T}$ is a set of labels, with $\tau \in \mathcal{T}$, and $\lambda : T \rightarrow \mathcal{T}$ is function that assigns labels to transitions.

If $\lambda(t) = \tau$, t is a *τ-transition*; otherwise, t is *observable*.

By translating the process model in Figure 1a, one can get the labeled net depicted in Figure 3a. The τ-transitions had to be added to receive the correct split and join behavior. Figure 3b shows the unfolding of the net in Figure 3a. Unfoldings and the respective algorithms are described in detail by Esparza and Heljanko in [EH08]. Loosely speaking, the unfolding is a more "simple" representation of the original net. Every *XOR-split* in the original net creates new branches in the unfolding that are never joined. This may lead to duplication of transitions and places, since everything that follows after a corresponding *XOR-join* in the original net is added to each created branch in the unfolding. As a result, the unfolding may have a high degree of redundancy, but its structural complexity is reduced compared to the original model. For details we refer the reader to [EH08].

Unfoldings can be large in size, possibly infinite, as compared to their originative nets. To make the unfolding feasible for application in practice, there exists the concept of the complete prefix unfolding, i.e., the minimum part of the unfolding that is capable of representing all valid markings of the original net. One needs to make a cut in the unfolding if a marking is reached the second time. The transition that leads to this marking a second time is called a *cutoff transition*. To illustrate this idea, Figure 3b shows the complete prefix unfolding of the Petri net in Figure 3a. While unfolding the net, the initial *XOR-split* opens two branches. Later in the unfolding, both these branches reach the *AND-splits*. This results in the labeled places p, q, s, and t. As one can see p and s refer to the same place a in the original net; q and t refer to b, respectively. As either p and s, or q and t

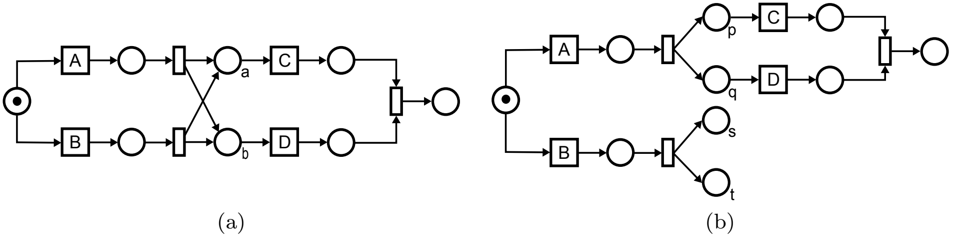

Figure 3: (a) A Petri net for the model in Figure 1a, and (b) its complete prefix unfolding

are marked, they both represent the same marking of a and b in the original net. Therefore, one of the two τ-transitions is a *cutoff transition*. Since we make a cut, it suffices to add the following transitions C and D to only one branch. This results in the complete prefix unfolding that is shown in Figure 3b.

2.3 Ordering Relations and the Modular Decomposition Tree

Ordering relations characterize the behavioral dependencies between the activities of a process model. Each pair of activities is either in *precedence*, *conflict*, or *concurrency* relation. These relations are defined as follows.

Definition 2.4 (Ordering relations)
Let $N = (P, T, F)$ be a complete prefix unfolding, let $x, y \in T$ be transitions of N.

- x *precedes* y ($x \rightsquigarrow y$), iff $(x, y) \in F^+$.
- x and y are in *conflict* ($x \# y$), iff
  $\exists\ t_1, t_2 \in T, t_1 \neq t_2 : (\bullet t_1 \cap \bullet t_2 \neq \emptyset) \wedge (t_1 \rightsquigarrow x) \wedge (t_2 \rightsquigarrow y)$.
- x and y are *concurrent* ($x \parallel y$), iff they are neither in precedence, nor in conflict.

The set $\mathcal{R}_N = \{\rightsquigarrow_N, \#_N, \|_N\}$ forms the ordering relations of N. As one can see from Definition 2.4, an activity precedes another activity, if and only if there exists a path of edges from one activity to the other. Two activities are in conflict, if one can find a place that is input to two different transitions each of them leading to one of the two activities. A pair of activities is considered concurrent in all other cases. For instance in Figure 3b, one can identify that transitions A and B are in conflict, since the initial place is input to both transitions. Transition A precedes transitions C and D, since we know that places s and t are outputs of a cutoff transition, it also follows that $B \rightsquigarrow C$ and $B \rightsquigarrow D$. C and D are concurrent. For details on the algorithm for computing ordering relations in a complete prefix unfolding, please refer to [PGD10].

Definition 2.5 (Ordering relations graph)
Let $N = (P, T, F, \mathcal{T}, \lambda)$ be a labeled net and let U be a complete prefix unfolding of N. The ordering relations graph of N is a triple $G_\lambda = (V, E, \mathcal{L})$, where V is the set of observable transitions of N, $\mathcal{L} = \{\epsilon, \rightsquigarrow, \#, \|\}$ is the set of labels, or *colors*,

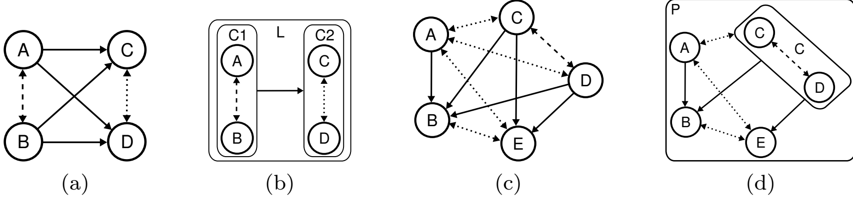

Figure 4: (a) The ordering relations graph of the complete prefix unfolding in Figure 3b, and (b) its modular decomposition tree; (c) the ordering relations of the complete prefix unfolding of a process model in Figure 2, and (d) its modular decomposition tree

and $E = V \times V \to \mathcal{L}$ is an edge labeling function, such that $E(x, y) = \oplus$, with $x, y \in V$ and $\oplus \in \mathcal{L} \backslash \epsilon$, if $x \oplus_N y$; otherwise $E(x, y) = \epsilon$.

From Definition 2.4 and Definition 2.5 it follows that between any pair of nodes in G_λ there is an edge whose color is not ϵ. For that reason, we say that $E(x, y) = \epsilon$ implies that there is no edge in the graph. Nevertheless, G_λ is connected, i.e., every pair of nodes is connected by at least one edge. Furthermore, we can identify, that $\|$ and $\#$ are symmetric relations, whereas $\rightsquigarrow$ and ϵ are asymmetric relations. In addition, $E(x, y) = \rightsquigarrow$ implies $E(y, x) = \epsilon$.

Figure 4a depicts the ordering relations graph of the process model in Figure 1a with its corresponding complete prefix unfolding in Figure 3b. The solid unidirectional arcs represent the *precedence* relation, dashed bidirectional arcs denote *exclusiveness*, and dotted bidirectional arcs represent *concurrency*.

An ordering relations graph is in fact a 2-structure, cf. [EGMS94]. In the following, we apply the theory of 2-structures to ordering relations graphs. Let $\mathcal{G} = (V, E, \mathcal{L})$ be an ordering relations graph. A *clan* of $\mathcal{G}$ is a subset $C \subseteq V$ of nodes of $\mathcal{G}$, such that $\forall x, y \in C\ \forall z \in V \backslash C : E(x, z) = E(y, z) \wedge E(z, x) = E(y, z)$, i.e., all nodes of the clan have uniform relations to all other nodes outside the clan. Obviously, V, $\emptyset$, and all $V' \subseteq V$ with $|V'| = 1$ are clans. Such clans are the trivial clans of $\mathcal{G}$. A *clan* C of $\mathcal{G}$ is a *prime* clan, if for all other clans C' of $\mathcal{G}$, either $C \subseteq C'$, $C' \subseteq C$, or $C \cap C' = \emptyset$.

Definition 2.6 (Primitive, Complete, Linear)
Let $\mathcal{C}$ be a clan of an ordering relations graph $\mathcal{G} = (V, E, \mathcal{L})$.

- $\mathcal{C}$ is *complete* (C), iff $\exists \oplus \in \{\#, \|\}\ \forall\ x, y \in \mathcal{C}, x \neq y : E(x, y) = \oplus$, i.e., all nodes in $\mathcal{C}$ are connected by edges of the same color. If $\oplus = \#$, then $\mathcal{C}$ is *XOR*-complete; otherwise $\mathcal{C}$ is *AND*-complete.
- $\mathcal{C}$ is *linear* (L), iff there exists a linear order $(x_1, \ldots, x_{|\mathcal{C}|})$ of elements of $\mathcal{C}$, such that $E(x_i, x_j) = \rightsquigarrow$ if $i < j$ and $E(x_i, x_j) = \epsilon$ otherwise.
- $\mathcal{C}$ is *primitive* (P), iff $\mathcal{C}$ is neither complete nor linear.

Note that a clan with less than three nodes cannot be primitive. Since prime clans are either contained within other prime clans or are disjoint, the set of all prime clans of an ordering relations graph forms a hierarchical decomposition.

Definition 2.7 (Modular Decomposition Tree (MDT))
The *Modular Decomposition Tree* (MDT) of an ordering relations graph $\mathcal{G}$, denoted by $MDT(\mathcal{G})$, is the containment hierarchy of all prime clans of $\mathcal{G}$.

Figure 4b shows the MDT of the ordering relations graph in Figure 4a. The MDT consists of two *complete* clans $C1$ and $C2$, where $C1$ is XOR-complete and $C2$ is AND-complete. $C1$ and $C2$ are contained in the linear clan L, which is also the root of the modular decomposition tree.

In [PGD10], the authors discovered relation between prime clans of the MDT and SESE regions of the RPST. Let $\mathcal{C}$ be a prime clan of the MDT and $\mathcal{D}$ its corresponding SESE region in the RPST.

- If $\mathcal{C}$ is linear, then there exists $\mathcal{D}$ that is trivial or polygon.
- If $\mathcal{C}$ is AND- (XOR-)complete, then there exists $\mathcal{D}$ that is an AND (XOR) bond.

Algorithm 1 Restructure an Acyclic Process Model, cf. [PGD10]

Input: An acyclic process model m
$n := process_model_to_petri_net(m)$
$u := compute_complete_prefix_unfolding(n)$
$org := get_ordering_relations_graph(u)$
$mdt := compute_modular_decomposition(org)$
`// traverse the MDT and output the RPST, if possible`
for all prime clan c in mdt in postorder **do**
  if c is primitive **then**
    FAIL
  else
    $generate_structured_sese_region(c)$
  end if
end for
return the RPST

Using these relations, one can employ the modular decomposition tree to synthesize a well-structured process model. One needs to perform a stepwise construction of the RPST, where each prime clan of the MDT results in the corresponding SESE region in the RPST. Following this approach for the MDT in Figure 4b, we receive the well-structured process model in Figure 1b. The complete clans $C1$ and $C2$ are replaced by their corresponding AND and XOR bonds containing the activities belonging to the trivial clans that are contained in $C1$ and $C2$. The polygon P represents the linear clan L. The start and end event are added for convenience.

The complete algorithm for structuring a process model, as described in [PGD10], is depicted in Algorithm 1. Algorithm 1 returns a result only if a process model can be well-structured completely and fails otherwise, even if the original process model can be partially structured.

3 Extension of the Structuring Algorithm

In this section, we aim at increase in the coverage of Algorithm 1, i.e., we allow partial structuring of a model when it has no well-structured representation. To increase the coverage, an extension of the algorithm must be able to handle primitive clans. As a running example in this section, we refer to the process model depicted in Figure 2. The RPST of this process model contains a rigid region. Computing the ordering relations graph of this rigid region results in the graph shown in Figure 4c, whereas Figure 4d shows its MDT. If one would run Algorithm 1 with the process model in Figure 2 as input, the algorithm will fail. Observe, however, that the MDT of the ordering relations graph contains a complete clan C that can be replaced by a bond in the RPST.

In the following, we provide an extension of Algorithm 1 that is able to transform the MDTs as in Figure 4d into process models. The next section gives a classification of primitive clans based on the colors of edges they contain. Furthermore, the next section investigates whether all of the classified classes of primitive clans must be considered when synthesizing process models. Afterwards, Section 3.2 shows an extension of the algorithm for handling one class of primitive clans.

3.1 Classification of Primitivity

In this section, we examine primitive clans and classify them according to the colors of edges they contain.

Let $N = (P, T, F)$ be a complete prefix unfolding. For all $x, y, z \in P \cup T$ holds:

1. $x \rightsquigarrow y \wedge y \rightsquigarrow z \Rightarrow x \rightsquigarrow z$, i.e., the *precedence* relation is transitive.
2. $x \# y \wedge y \rightsquigarrow z \Rightarrow x \# z$, i.e., the *conflict* relation is inherited by the *precedence* relation.

A primitive clan $\mathcal{C}$ is *valid* if it obeys these two properties; otherwise $\mathcal{C}$ is *invalid.* For the transformation of a primitive clan into a process model we only consider *valid* primitive clans, since the behavioral dependencies described by an *invalid* primitive clan might be contradicting and, therefore, are unrealizable without the introduction of additional variables or may lead to unsound process models.

Definition 3.1 ($\rightsquigarrow$-, AND-, XOR-, Mixed-primitive)
Let $\mathcal{C}$ be a primitive clan.

1. $\mathcal{C}$ is a $\rightsquigarrow$-primitive, or precede-primitive, iff $\forall\, x, y \in \mathcal{C}, x \neq y : (E(x,y) = \rightsquigarrow \wedge E(y,x) = \epsilon) \vee (E(x,y) = \epsilon \wedge E(y,x) = \rightsquigarrow)$, i.e., between the nodes in $\mathcal{C}$ there are only precedence relations.
2. $\mathcal{C}$ is an *AND*-primitive, iff $\forall\, x, y \in \mathcal{C}, x \neq y : (E(x,y) \in \{\|, \rightsquigarrow\} \vee E(y,x) \in \{\|, \rightsquigarrow\}) \wedge E(x,y) \neq \# \wedge E(y,x) \neq \#$, i.e., between the nodes in $\mathcal{C}$ there are only precedence and concurrency relations.
3. $\mathcal{C}$ is a *XOR*-primitive, iff $\forall\, x, y \in \mathcal{C}, x \neq y : (E(x,y) \in \{\#, \rightsquigarrow\} \vee E(y,x) \in \{\#, \rightsquigarrow\}) \wedge E(x,y) \neq \| \wedge E(y,x) \neq \|$, i.e., between the nodes in $\mathcal{C}$ there are only precedence and conflict relations.
4. $\mathcal{C}$ is a Mixed-primitive, iff it has edges of all colors.

We now check these classes for validity. As Proposition 3.1 suggests, there exist no valid $\rightsquigarrow$-primitives or *XOR*-prmitives.

Proposition 3.1 *Let P be a primitive clan.*

1. *If P is a $\rightsquigarrow$-primitive, then P is invalid.*
2. *If P is an XOR-primitive, then P is invalid.*

In the following, we sketch ideas for the proof of Proposition 3.1.

1. Let P be valid. Then, it holds that $\forall\ x, y, z \in P : x \rightsquigarrow y \wedge y \rightsquigarrow z \Rightarrow x \rightsquigarrow z$. Let $succ(x) = \{y \mid x, y \in P, E(x, y) = \rightsquigarrow\}$ denote the set of successors of node x. Intuitively, since there are no other relations than precedence in the corresponding unfolding, all transitions are located on one path connecting them all in a sequence. Because of this sequential ordering it holds that $\forall\ x, y \in P, x \neq y : |succ(x)| \neq |succ(y)|$. According to Definition 3.1 it follows that P is linear, which contradicts the fact that P is primitive.
2. The basic case is a primitive P that contains three nodes. Again, we assume that P is valid. We distinguish two cases.
   Case 1: $\forall\ x, y \in P, x \neq y : E(x, y) = \#$, i.e., we solely find conflict relations. As all edges are of the same color, this implies that P is *XOR*-complete and, therefore, not primitive.
   Case 2: $\exists\ x, y, z \in P, x \neq y, y \neq z, x \neq z$, such that $E(x, y) = \#$ and $E(x, z) = \rightsquigarrow$, i.e., we observe both relations in P. From validity then follows that $E(y, z) = \#$. Obviously, the set $\{x, z\}$ is a linear clan L. L and the trivial clan $\{y\}$ form a *XOR*-complete clan. Therefore, P is not primitive. Thus, the proposition holds for the base case.
   In each step we add a new node w to P. There are four cases:
   Case 1: $\forall\ x \in P, x \neq w : E(x, w) = \#$, i.e., w is in conflict to all other nodes in P. From the base step we know that $P \backslash \{w\}$ is not primitive. Therefore, P is either linear or *XOR*-complete. Inserting w in conflict to all other nodes is equivalent to adding a new trivial clan $\{w\}$ to P. If P was linear before, then the resulting clan P' is *XOR*-complete containing the old linear P and the trivial $\{w\}$. If P was *XOR*-complete, we simply add $\{w\}$, and P is still *XOR*-complete. Therefore, P is not primitive.
   Case 2: $\forall\ x \in P, x \neq w : E(x, w) = \rightsquigarrow$. This implies that P and $\{w\}$ are in a linear order. Therefore, $P' = P \cup \{w\}$ is linear and not primitive.
   Case 3: $\forall\ x \in P, x \neq w : E(x, w) = \epsilon$. This is analogous to Case 2.
   Case 4: $\exists\ x, y, z \in P, x \neq y, y \neq w, x \neq w, z \neq x$, such that $E(x, y) = \#$, $E(x, w) = \rightsquigarrow$, and $E(w, z) = \#$, i.e., w must fulfill relations of both types. From validity it follows that $\forall\ u \in P : E(u, x) = \# \Rightarrow E(u, w) = \#$, i.e., w is in conflict to everything x is in conflict with. Furthermore, from the transitivity of $\rightsquigarrow$ we know that $\forall\ v \in P : E(v, x) = \rightsquigarrow \Rightarrow E(v, w) = \rightsquigarrow$. This is equivalent to adding the trivial clan $\{w\}$ to the linear containing x, which itself is nested in a *XOR*-complete clan. Therefore, the structure of P does not change and is still not primitive.

Hence, it suffices to investigate *AND*- and Mixed-primitives. The following section provides an algorithm for synthesizing process models from *AND*-primitives.

3.2 Handling AND-Primitives

AND-primitives form a simple class of primitivity. An example of an *AND*-primitive is shown in Figure 4d, which represents the modular decomposition tree of our running example. The exclusiveness relation between C and D is not part of the primitive, since it is "hidden" inside of a *XOR*-complete clan. Algorithm 2 depicts the essentials for synthesizing a process model from an *AND*-primitive. We define the following auxiliary functions:

- `new_node(type)`: Create a node of the given type.
- `set_label(node, l)`: Set the label of a node.
- `label(n)`: Get the label of node `n`.
- `new_edge(t1, t2)`: Create an edge from `t1` to `t2`.
- `set_target(edge, node)`: Set the target node of an edge.
- `set_source(edge, node)`: Set the source node of an edge.
- `add(model, el)`: Add an element to a process model.

In the first step, the algorithm creates a task for each node of the primitive; this includes potential completes and linear clans. At the second step, we add control flows between the tasks, where the corresponding nodes in P are in precedence and this precedence is not covered by any transitivity. Lines 17 and 18 in Algorithm 2 add an initial *AND*-split and a final *AND*-join as a start and end node of the model. The last step adds further splits and joins where required, and connects tasks with no incoming (outgoing) edges with the initial *AND*-split (the final *AND*-join). Furthermore, tasks that correspond to completes or linear clans are replaced by their respective structured representations, as computed by Algorithm 1.

Figure 5 illustrates the algorithm for our running example. Figure 5a represents the result after steps 1 and 2, whereas Figure 5b shows the final result. As one can see, task *CD* has been replaced by a *XOR* block containing the two activities, and *AND*-joins (-splits) have been added where a task had multiple incoming (outgoing) control flow edges. In fact, Figure 5b depicts the maximally structured representation for our example process model.

Summarizing the findings of this section, we observe that in the case of a clear separation of concurrency and conflict relations the algorithm returns the maximally structured representation of a given process model.

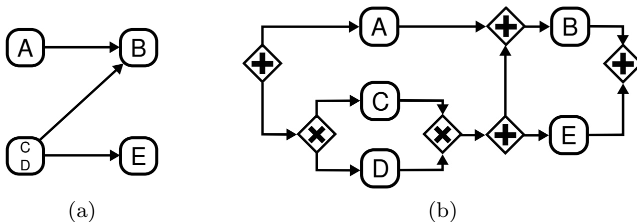

Figure 5: (a) Result after Step 1 and 2 of Algorithm 2, and (b) the final result for our running example in Figure 2

Algorithm 2 Synthesis of a Process Model from an *AND*-primitive

Input: An *AND*-primitive P
Output: A process model m

$m :=$ an empty process model
`// Step 1`
5: **for all** node $n \in P$ **do**
  $t := new_node(Task);\ set_label(label(n));\ add(m, t);$
end for

`// Step 2`
for all nodes $n_1, n_2 \in P$ where $E(n_1, n_2) = \rightsquigarrow$ **do**
10: **if** $\exists\ x \in P : E(n_1, x) = \rightsquigarrow \wedge E(x, n_2) = \rightsquigarrow$ **then**
    continue
  end if
  $t_1 :=$ task with label $label(n_1)$
  $t_2 :=$ task with label $label(n_2)$
15: $e := new_edge(t_1, t_2);\ add(m, e);$
end for

$as_0 := new_node(ANDsplit);\ add(m, as_0);$
$aj_0 := new_node(ANDjoin);\ add(m, aj_0);$

`// Step 3`
20: **for all** task t in m **do**
  if t has multiple outgoing edges **then**
    $as := new_node(ANDsplit)$
    for all outgoing edge e from t **do**
      $set_source(e, as)$
25:    **end for**
    $e = new_edge(t, as);\ add(m, as);\ add(m, e);$
  else if t has no outgoing edge **then**
    $e = new_edge(as_0, t);\ add(m, e);$
  end if
30: **if** t has multiple incoming edges **then**
    $aj := new_node(ANDjoin)$
    for all incoming edge e to t **do**
      $set_target(e, aj)$
    end for
35:    $e = new_edge(aj, t);\ add(m, aj);\ add(m, e);$
  else if t has no outgoing edge **then**
    $e = new_edge(t, aj_0);\ add(m, e);$
  end if
  if t corresponds to a complete or linear in P **then**
40:    replace t with corresponding block or polygon
  end if
end for
return m

4 Conclusion

In this paper, we have investigated primitive clans of the modular decomposition tree to achieve maximally structured representations of inherently unstructured acyclic sound process models. We have distinguished the primitive clans according to the ordering relations contained in them. In the first step, we argued that not all types of primitives are valid for synthesizing a process model. In the second step, we presented Algorithm 2 to synthesize a process model for the case of *AND*-primitives. This algorithm has been implemented and integrated into the existing approach presented in [PGD10], to avoid failing for that type of primitivity. The second valid primitive class, Mixed-primitives, requires further investigation due to its high complexity. Furthermore, future work should focus on lifting the acyclic restriction imposed on process models.

References

[DDO07] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Formal Semantics and Analysis of BPMN Process Models using Petri Nets, 2007.

[EGMS94] Andrzej Ehrenfeucht, Harold N. Gabow, Ross M. McConnell, and Stephen J. Sullivan. An O(n2) divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *J. Algorithms*, 16(2):283–294, 1994.

[EH08] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, March 2008.

[KtHB00] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On Structured Workflow Modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 431–445, London, UK, 2000. Springer-Verlag.

[MRvdA10] Jan Mendling, Hajo A. Reijers, and Wil M. P. van der Aalst. Seven process modeling guidelines (7PMG). *Information & Software Technology*, 52(2):127–136, 2010.

[PGD10] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring Acyclic Process Models. In *Proceedings of the 8th International Conference on Business Process Management (BPM)*, Hoboken, NJ, US, 2010.

[PVV10] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified Computation and Generalization of the Refined Process Structure Tree. In *Proceedings of the 7th International Workshop on Web Services and Formal Methods (WS-FM)*, Hoboken, NJ, US, 2010.

[VVK08] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The Refined Process Structure Tree. In *Proceedings of the 6th International Conference on Business Process Management (BPM)*, Berlin, Heidelberg, 2008. Springer-Verlag.

[VVK09] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The Refined Process Structure Tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.