Structuring Acyclic Process Models

Artem Polyvyanyy[a,*], Luciano García-Bañuelos[b], Marlon Dumas[b]

[a] *Hasso-Plattner-Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany*
[b] *Institute of Computer Science, University of Tartu, J. Liivi 2, Tartu 50409, Estonia*

Abstract

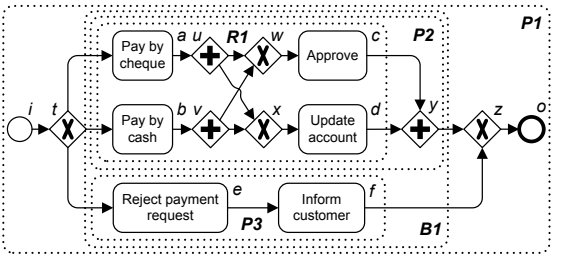
This article studies the problem of transforming a process model with an arbitrary topology into an equivalent well-structured process model. While this problem has received significant attention, there is still no full characterization of the class of unstructured process models that can be transformed into well-structured ones, nor an automated method for structuring any process model that belongs to this class. This article fills this gap in the context of acyclic process models. The article defines a necessary and sufficient condition for an unstructured acyclic process model to have an equivalent well-structured process model under fully concurrent bisimulation, as well as a complete structuring method. The method has been implemented as a tool that takes process models captured in the BPMN and EPC notations as input. The article also reports on an empirical evaluation of the structuring method using a repository of process models from commercial practice.

Keywords: Process modeling, Structured modeling, Structuring, Model equivalence, Petri net unfolding, Modular decomposition

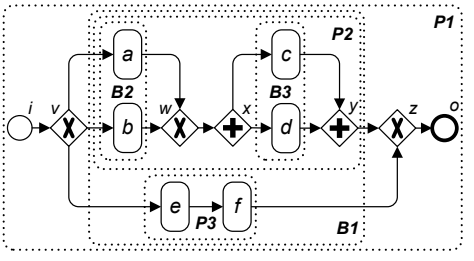
1. Introduction

In contemporary business process modeling notations, such as the Business Process Model and Notation (BPMN) [1] and Event-driven Process Chain (EPC) [2], a process model is composed of nodes (e.g., tasks, events, gateways) connected by a "flow" relation. Although these notations allow process models to have almost any topology, it is often desirable that models abide by some structural rules. In this respect, a well-known property of process models is that of *(well-)structuredness* [3], meaning that for every node with multiple outgoing arcs (a *split*) there is a corresponding node with multiple incoming arcs (a *join*), and vice versa, such that the set of nodes between the split and

*Corresponding author
Email addresses: artem.polyvyanyy@hpi.uni-potsdam.de (Artem Polyvyanyy), luciano.garcia@ut.ee (Luciano García-Bañuelos), marlon.dumas@ut.ee (Marlon Dumas)



(a)



(b)

Figure 1: (a) An unstructured acyclic process model, and (b) its equivalent structured version

the join forms a single-entry-single-exit (SESE) region; otherwise the process model is *unstructured*. For example, Figure 1(a) shows an unstructured process model (splits u, v and joins w, x do not have a corresponding node), while Figure 1(b) shows an equivalent structured model. The models are captured using BPMN language. Figure 1(b) uses short names for tasks $(a, b, c, \ldots)$, which appear next to each task in Figure 1(a). Observe that the equivalent structured model captures the same information about potential concurrent executions of tasks as the unstructured model. As will be explained later, such a relation on process models corresponds to the notion of fully concurrent bisimulation [4].

This article studies the problem of automatically transforming unstructured process models into equivalent well-structured models. The motivations for such a transformation are manifold. Firstly, it has been empirically shown that structured process models are easier to comprehend and less error-prone than unstructured ones [5]. Similarly, it has been shown in [6, 7] that structuredness and modularity (e.g., by usage of procedures) improves software maintainability. Thus, a transformation from an unstructured into a structured process model can be used as a refactoring technique for increasing process model understandability. Secondly, a number of existing process model analysis techniques only work for structured models. For example, a method for calculating cycle time and capacity requirements of structured process models is outlined in [8], while a method for analyzing time constraints in structured process models is presented in [9]. By transforming unstructured process models into structured ones, we can extend the applicability of these techniques to a larger class of models. Thirdly, a transformation from unstructured to structured process models can be used

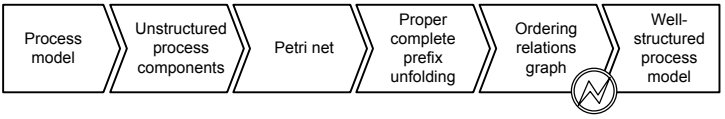


Figure 2: Overview of the proposed structuring method

to implement converters from graph-oriented process modeling languages to structured process modeling languages, e.g., from BPMN to BPEL [10].

In the context of flowcharts without parallel splits and joins it has been shown that any unstructured flowchart can be transformed into a structured one [11][1]. If we add parallel splits and joins, this result no longer holds: There exist unstructured process models that do not have equivalent structured ones [3]. Several authors have attempted to classify the sources of unstructuredness in process models [12–14] and to define automated methods for structuring process models [15–17]. However, these methods are incomplete: There is currently no full characterization of the class of *inherently* unstructured process models, i.e., unstructured process models that have no equivalent structured model. Also, none of the existing structuring methods is complete. In fact, this problem has not been fully solved even for acyclic process models. This article fills this gap.

To streamline the presentation, we make several assumptions. Firstly, we consider process models to be composed of nodes (tasks, events, gateways) and control flow relations. In terms of BPMN, this means that we abstract away from other process model elements such as artifacts, annotations, associations, groups, pools, lanes, message flows, sub-process invocations and attributes associated with sub-process invocations, e.g., repetition. Nonetheless, the proposed method is applicable even if these types of elements are present in the input model. Simply put, these ancillary elements and attributes need to be moved along with the tasks or events to which they are attached. In the same vein, we do not distinguish between events and tasks since, for the purpose of the transformation, both of them can be treated equally. Secondly, we consider only sound process models [18]. This restriction is natural since soundness is a widely-accepted correctness criterion for process models. Thirdly, we consider process models in which every node has only one incoming or one outgoing arc. This restriction is merely syntactical because one can trivially split a node with multiple incoming and multiple outgoing arcs into two nodes: one node with a single outgoing arc and the other with a single incoming arc. Finally, we do not deal with the following BPMN constructs: *or* gateways, complex gateways, error events and non-interrupting events. Lifting this latter restriction is left as future work.

Given this setting, the main contribution of the article is a method for transforming an unstructured *acyclic* process model into an equivalent structured one whenever such transformation is possible. The main steps of the method are summarized in Figure 2. The process model is first decomposed into a

[1]In the case of multi-exit cycles, this transformation requires the use of "break" statements or boolean variables in the transformed model.

"process structure tree": A hierarchy of so-called *process components*, where each component corresponds to a SESE region. Each component can be seen as a process model by itself and can be classified either as well-structured or unstructured. The aim of the method is to transform every unstructured component into an equivalent structured one, when possible. To this end, the process component is first translated into a Petri net [19]. A technique known as *Petri net unfolding* is then employed in order to discover the elementary ordering relations that exist between tasks in the component. These ordering relations are analyzed using a technique known as *modular decomposition*, which extracts the fundamental "modules" that exist in the ordering relations graph. We demonstrate that if the resulting modules are all of certain types, the process component can be transformed into an equivalent structured one. Otherwise, the process component is inherently unstructured.

This article is an extended and revised version of an earlier conference paper [20]. The main enhancements with respect to the conference paper are the ability to deal with process models with multiple source and multiple sink nodes, and an empirical evaluation of the structuring method using process models taken from industrial practice.

The remainder of the article is structured as follows: The next section defines the notions of process model and refined process structured tree and reviews related work. Next, Section 3 introduces a semantics of process models based on Petri nets. Section 4 then introduces the behavioral equivalence used in this article, viz. fully concurrent bisimulation, and shows that two acyclic process models are equivalent under this equivalence notion if and only if they have the same set of ordering relations. This result is used in Section 5 to characterize the class of acyclic process models that can be structured and to define a structuring algorithm. In Section 5, it is assumed that process models have a single source and a single sink node. This restriction is lifted in Section 6. Section 7 presents an empirical evaluation of the proposed structuring method on a process model collection taken from industrial practice. Finally, Section 8 concludes the article and outlines directions for future work.

2. Background and Related Work

In this section we introduce the notion of process model (Section 2.1) and a technique for decomposing process models into process components (Section 2.2). Afterwards, in Section 2.3, we analyze related work with respect to their ability to structure different types of process components.

2.1. Process Models

As discussed in the introduction, we consider process models consisting of tasks and gateways, as captured in the following definition.

Definition 2.1 (Process model).
A *process model* is a tuple $W = (A, G_{\oplus}, G_{\otimes}, C, \mathcal{A}, \mu)$, where A is a non-empty set of *tasks*, or *activities*, $G_{\oplus}$ is a set of *and gateways*, $G_{\otimes}$ is a set of *xor gateways*

(these sets are disjoint). We write $G = G_{\oplus} \cup G_{\otimes}$ for all *gateways* and $V = A \cup G$ for all *nodes* of the process model. $C \subseteq V \times V$ defines the *control flow*. $\mathcal{A}$, $\tau \in A$, is a set of *names* and $\mu : A \to \mathcal{A}$ is a function that assigns each task a name.

A node $v \in V$ is a *source* node, if $\bullet v = \varnothing$, and it is a *sink* node, if $v\bullet = \varnothing$, where $\bullet v$ stands for a set of immediate predecessors and $v\bullet$ stands for a set of immediate successors of node v. As mentioned in the introduction, we assume that process models meet certain structural requirements. Every task $a \in A$ has at most one incoming and at most one outgoing arc, i.e., $|\bullet a| \leq 1 \wedge |a\bullet| \leq 1$, while each gateway is either a split or a join: A gateway $g \in G$ is a *split*, if $|\bullet g| = 1 \wedge |g\bullet| > 1$. A gateway $g \in G$ is a *join*, if $|\bullet g| > 1 \wedge |g\bullet| = 1$. Source and sink nodes are tasks and every node is on a path from some source task to some sink task.

We employ a notation similar to BPMN for visualization of process models. Figure 1(a) shows a process model which contains eight tasks, i.e., $A = \{i, a, b, c, d, e, f, o\}$. Note that i and o are *silent* tasks, i.e., $\mu(i) = \tau = \mu(o)$. We visualize silent tasks as start, intermediate, or end events in BPMN. Note that we use silent tasks for technical purposes only, e.g., representation of source and sink tasks. An observable task is drawn as a rectangle that has rounded corners with its name inside. Gateways are visualized as diamonds: Gateways of type *xor*, or *exclusive* gateways, use a marker which is shaped like an "×" inside the diamond shape. Gateways of type *and*, or *parallel* gateways, use a marker which is shaped like a "+" inside the diamond shape. The set $\{t, u, v, w, x, y, z\}$ is the set of gateways of the process model in Figure 1(a). Gateways t, w, x, and z are exclusive, while gateways u, v, and y are parallel. Gateways t, u, and v are splits, while gateways w, x, y, and z are joins. Finally, control flow arcs are drawn as directed edges between the nodes of the process model.

2.2. Process Components

The starting point of the proposed structuring method is a parsing technique for process models presented in [21] (originally proposed in [22]). This parsing technique decomposes a process model into SESE regions, hereby called *process components*. In other words, a process component is a subset of arcs of a process model such that the subgraph induced by these arcs has a single entry node and a single exit node. These two nodes are called *boundary nodes* as they connect the component with the rest of the model. A process component is *canonical*, if it does not overlap (on the set of arcs) with any other process component, meaning that any two canonical process components are either disjoint or one is contained in the other. Canonical process components naturally form a hierarchy, leading to the following definition.

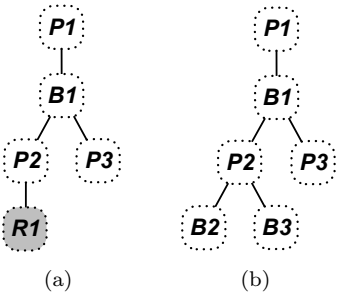


Figure 3: The (simplified) RPSTs of process models in Figure 1

Definition 2.2 (Refined process structure tree).
The *refined process structure tree* (*RPST*) of a process model is the set of all its canonical process components.

The RPST of a given process model is unique and can be computed in linear time [21]. Figure 1(a) and Figure 1(b) show the RPSTs in the form of dotted boxes superposed on the process models. For the sake of presentation, the same RPSTs are depicted as trees in Figure 3. Every region inside a dotted box defines a canonical process component, which is composed of arcs that are inside or intersect the region. For instance, component $B1$ in Figure 1(a) has two boundary nodes: t and z. Node t is the *entry* and node z is the *exit* of the component. According to [21–23], every canonical process component belongs to one out of four structural classes.

Definition 2.3 (Trivial, Polygon, Bond, Rigid).
Let C be a process component of a process model.

- C is a *trivial* component, iff C is singleton, i.e., C contains a single arc.
- C is a *polygon* component, iff there exists a sequence $(r_0, \ldots, r_n)$, $n \in \mathbb{N}$, of canonical components of the process model, such that $C = \bigcup_{i=0}^{i=n} r_i$, the entry of C is the entry of r_0, the exit of C is the exit of r_n, and the exit of r_j is the entry of r_{j+1}, $0 \leq j < n$.
- C is a *bond* component, iff there exists a set R of canonical components of the process model, such that $C = \bigcup_{r \in R} r$ and every component in R has the same boundary nodes as C.
- C is a *rigid* component, iff C is neither a trivial, nor a polygon, nor a bond component.

For instance in Figure 1(a), polygon $P1$ is the root of the RPST and corresponds to the whole process model. Polygon $P1$ is composed of bond $B1$ and two trivial components $\{(i,t)\}$ and $\{(z,o)\}$. In turn, bond $B1$ contains polygons $P2$ and $P3$. Observe that trivial components and polygons that are composed of two trivial components are not explicitly visualized for simplicity reasons – neither in Figure 1, nor in Figure 3. Note also that names of components hint at their structural class, e.g., $P1$ is a polygon, $B1$ is a bond, and $R1$ is a rigid component.

The RPST provides a basis for defining the notion of well-structuredness.

Definition 2.4 (Well-structured process model).
A process model is *(well-)structured*, if and only if its RPST contains no rigid process component; otherwise the process model is *unstructured*.

In a process model that satisfies the condition in the above definition, every split has a corresponding join so that the region between the split and the join is a SESE region, and vice versa (by virtue of the definition of a trivial, polygon, and bond component). Every rigid process component contains a node, either a split or a join, which has no corresponding node to define a SESE region.

The process model in Figure 1(a) contains rigid $R1$ and is, therefore, unstructured. On the other hand, the process model in Figure 1(b) is well-structured –

its RPST contains no rigid process component. Hence, if we had a method for transforming every rigid component in the RPST into an equivalent structured component, we would be able to transform any process model into a structured model by traversing the RPST bottom-up and replacing each rigid by its equivalent structured component. In the running example, this corresponds to replacing $R1$ (highlighted with grey background in Figure 3(a)) with components $B2$ and $B3$, as hinted in Figure 3(b). Accordingly, the rest of the article focuses on how to structure rigid components.

Existing methods for structuring rigid components differ depending on the types of gateways present in the rigid and whether the rigid contains cycles or not. In this respect, it is useful to further classify rigid components as follows: A homogeneous rigid contains either only *xor* or only *and* gateways. We call these rigids *(homogeneous) xor rigids* and *(homogeneous) and rigids*, respectively. A heterogeneous rigid contains a mixture of *and*/*xor* gateways. Heterogeneous and homogeneous *xor* rigids are further classified into cyclic, if they contain at least one cyclic path, or acyclic. Importantly, we do not classify homogeneous *and* rigids as cyclic or acyclic as process models with cyclic *and* rigids are unsound [24]. Given this background, a taxonomy of process components is provided in Figure 4.

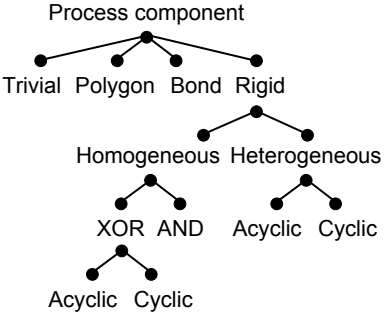


Figure 4: A taxonomy of components

2.3. Related Work

A large body of work on flowcharts and GOTO program transformation [11, 25] has addressed the problem of structuring *xor* rigids. In some cases, these transformations introduce additional boolean variables in order to encode part of the control flow, while in other cases they require certain nodes to be duplicated.

One of the earliest studies on the problem of structuring process models with concurrency is that of Kiepuszewski et al. [3]. The authors showed that not all acyclic *and* rigids can be structured by putting forward a counter-example, which is essentially the one presented in Figure 5. The authors showed that there exists no well-structured process model equivalent to this one under a behavioral equivalence notion which preserves the level of observed concurrency, viz. fully concurrent bisimulation, which we shall discuss later. However, this work does not provide a full characterization of the class of models that can be structured, nor it defines any automated structuring technique. Instead, some causes of unstructuredness are explored. In a similar vein, [12] presents a taxonomy of unstructuredness

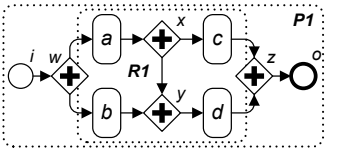


Figure 5: An inherently unstructured process model

in process models, covering cyclic and acyclic rigids. Again, the taxonomy is incomplete, i.e., it does not cover all possible cases of models that can be structured. Also, the authors do not define an automated structuring algorithm.

In [13], the authors outline a classification of process components using *region trees*. Region trees enable an incremental approach to the analysis and structuring of process models. However, the authors do not provide a complete structuring method for acyclic heterogeneous rigids, e.g., the one in Figure 1(a). Similar remarks apply to [26]. In [16], the authors propose a method for structuring cyclic *xor* rigids. In [15], the authors propose a method for structuring *xor* rigids based on GOTO program transformations, and extends this method to process graphs where *xor* rigids are nested inside bonds. Again, this method cannot deal neither with *and* rigids nor with heterogeneous rigids.

The problem of structuring process models is relevant in the context of designing translations between process definition languages, e.g., BPMN-to-BPEL transformations. BPMN-to-BPEL transformations, e.g., [17], treat rigid components as black-boxes which are translated using BPEL links or event handlers, rather than seeking to transform these rigids into structured components.

Recently, refactoring process models in large repositories has gained considerable attention [27, 28]. In that context, structuring is required, for instance, to reduce the complexity of process models containing redundant elements, such as superfluous control flow arcs, while preserving the original behavior. An empirical study reported in [29] revealed that significant size reductions and reuse can be achieved by refactoring duplicate components in process model repositories. It is also suggested that further refactoring opportunities could be uncovered only after structuring the models in these repositories.

3. Execution Semantics of Process Models

As outlined in Section 1, the proposed structuring method relies on a Petri net semantics of process models. For the sake of making the paper self-contained, we present below some standard definitions associated to Petri nets. We then introduce a mapping from process models to Petri nets.

Definition 3.1 (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.

We identify F with its characteristic function on the set $(P \times T) \cup (T \times P)$. For a node $x \in P \cup T$, $\bullet x = \{y \in P \cup T \mid F(y, x) = 1\}$ is a *preset*, whereas $x\bullet = \{y \in P \cup T \mid F(x, y) = 1\}$ is a *postset* of x. A node $x \in P \cup T$ is an *input* (*output*) node of a node $y \in P \cup T$, if $x \in \bullet y$ ($x \in y\bullet$). For $X \subseteq P \cup T$, $\bullet X = \bigcup_{x \in X} \bullet x$ and $X\bullet = \bigcup_{x \in X} x\bullet$. A place $p \in P$ is a *source* place, if $\bullet p = \varnothing$ and it is a *sink* place, if $p\bullet = \varnothing$. We denote by F^+ the transitive closure of F, and by F^* – the reflexive and transitive closure of F.

Nets have precise execution semantics defined in terms of a token game.

Definition 3.2 (Net semantics). Let $N = (P, T, F)$ be a net.

- $M : P \to \mathbb{N}_0$ is a *marking* of N assigning each place $p \in P$ a number $M(p)$ of *tokens*. $[p]$ denotes the marking where place p contains just one token and all other places contain no tokens. We identify M with the multiset containing $M(p)$ copies of p for every $p \in P$.
- For any transition $t \in T$ and for any marking M of N, t is *enabled* at M, denoted by $(N, M)[t\rangle$, iff $\forall p \in {\bullet t} : M(p) \geq 1$.
- If $t \in T$ is enabled at M, then it can *fire*, which leads to a new marking M', denoted by $(N, M)[t\rangle(N, M')$. The new marking M' is defined by $M'(p) = M(p) - F(p,t) + F(t,p)$, for each place $p \in P$.
- Let M_0 be a marking. If $(N, M_0)[t_1\rangle(N, M_1) \ldots (N, M_{n-1})[t_n\rangle(N, M_n)$ are transition firings, then a sequence of transitions $\sigma = (t_1, \ldots, t_n)$ is a *firing sequence* leading from M_0 to M_n.
- For any two markings M and M' of N, M' is *reachable* from M in N, denoted by $M' \in [N, M\rangle$, iff there exists a firing sequence σ leading from M to M'. Note that σ can be the empty sequence. We have $M \in [N, M\rangle$ for every M of N.
- A *net system*, or a *system*, is a pair (N, M_0), where N is a net and M_0 is a marking of N. M_0 is called the *initial marking* of N.

We expect all nets to be *T-restricted*, i.e., every transition of a net has at least one input place and at least one output place. Otherwise, we assume the natural completion of a net, i.e., the net gets modified so that every transition without an input (output) place gets a fresh input (output) place. By $Min(N)$ we denote the set of source places of net N. In the following, when we mention a net N in the context of a net system, we assume N with its natural marking, i.e., the marking comprising one token at each place in $Min(N)$ and no tokens elsewhere.

Workflow (WF-)nets are a subclass of Petri nets specifically designed to represent workflow procedures [30]. A WF-net is a net with two special places: one to mark the start and the other the end of a workflow execution.

Definition 3.3 (WF-net, Short-circuit net, WF-system).
A Petri net $N = (P, T, F)$ is a *workflow net*, or a *WF-net*, iff N has a dedicated source place $i \in P$, N has a dedicated sink place $o \in P$, and the *short-circuit net* $N^\star = (P, T \cup \{t^\star\}, F \cup \{(o, t^\star), (t^\star, i)\})$, $t^\star \notin T$, of N is strongly connected. A *WF-system* is a pair (N, M_i), where $M_i = [i]$.

Soundness and safeness are basic properties of WF-systems [30]. Soundness states that every execution of a WF-system ends with a token in the sink place, and once a token reaches the sink place, no other tokens remain in the net. Safeness refers to the fact that there is never more than one token in a place.

Definition 3.4 (Liveness, Safeness, Soundness).
- A system (N, M_0), $N = (P, T, F)$, is *live*, iff for every reachable marking $M \in [N, M_0\rangle$ and for every transition $t \in T$, there exists a marking $M' \in [N, M\rangle$, such that $(N, M')[t\rangle$.
- A system (N, M_0) is *bounded*, iff the set $[N, M_0\rangle$ is finite. A system (N, M_0), $N = (P, T, F)$, is *safe*, iff $\forall\ M \in [N, M_0\rangle\ \forall\ p \in P : M(p) \leq 1$.

- A WF-system (N, M_i) is *sound*, iff the short-circuit system $(N^\star, M_i)$ is live and bounded.

Petri nets have a great expressive power. They can be used to model a large variety of distributed systems. However, it is often sufficient to reduce investigations to a subclass of nets. In the following, we shall make extensive use of a structural subclass of free-choice nets [31, 32]. In a free-choice net two places that share an output transition may not have any other output transitions and two transitions that share an input place may not have any other input places.

Definition 3.5 (Free-choice net).
A net $N = (P, T, F)$ is *free-choice*, iff $\forall p \in P,\ |p\bullet| > 1 : \bullet(p\bullet) = \{p\}$.

It is often useful to distinguish between observable and silent transitions of a net. Accordingly, the notion of a net must be extended.

Definition 3.6 (Labeled net).
A *labeled* net is a tuple $N = (P, T, F, \mathcal{T}, \lambda)$, where (P, T, F) is a net, $\mathcal{T}$ is a set of *labels*, such that $\tau \in \mathcal{T}$, and $\lambda : T \to \mathcal{T}$ is a function that assigns each transition a label. If $\lambda(t) \neq \tau$, then t is *observable*; otherwise, t is *silent*. λ is *distinctive*, if it is injective on the set of all observable transitions.

Observable transitions are designed to represent actions of the distributed system that are visible to the outside world, while silent transitions encode the internal actions of the system.

The execution semantics of process models is defined by means of a mapping to labeled free-choice Petri nets.

Definition 3.7 (WF-net of a process model).
Let $W = (A, G_{\oplus}, G_{\otimes}, C, \mathcal{A}, \mu)$ be a process model. Let I and O be source tasks and sink tasks of W, respectively. The labeled net $N = (P, T, F, \mathcal{T}, \lambda)$ that *corresponds* to W is defined by:

- $P = \{p_x \mid x \in G_{\otimes}\} \cup \{p_{x,y} \mid (x,y) \in C \wedge y \in A \cup G_{\oplus}\} \cup \{p_x \mid x \in I \cup O\}$.
- $T = \{t_x \mid x \in A \cup G_{\oplus}\} \cup \{t_{x,y} \mid (x,y) \in C \wedge x \in G_{\otimes}\}$.
- $F = \{(t_x, p_y) \mid (x,y) \in C \wedge x \in A \cup G_{\oplus} \wedge y \in G_{\otimes}\} \cup$
  $\{(t_x, p_{x,y}) \mid (x,y) \in C \wedge x, y \in A \cup G_{\oplus}\} \cup \{(t_{x,y}, p_y) \mid (x,y) \in C \wedge x, y \in G_{\otimes}\} \cup$
  $\{(t_{x,y}, p_{x,y}) \mid (x,y) \in C \wedge x \in G_{\otimes} \wedge y \in A \cup G_{\oplus}\} \cup$
  $\{(p_x, t_{x,y}) \mid (x,y) \in C \wedge x \in G_{\otimes}\} \cup \{(p_{x,y}, t_y) \mid (x,y) \in C \wedge y \in A \cup G_{\oplus}\} \cup$
  $\{(p_x, t_x) \mid x \in I\} \cup \{(t_x, p_x) \mid x \in O\}$.
- $\mathcal{T} = \mathcal{A}$, $\lambda(t_x) = \mu(x)$, $t_x \in T$, $x \in A$; otherwise $\lambda(t) = \tau, t \in T$.

For example, Figure 6 shows the net that corresponds to the process model in Figure 1(a). The figure highlights the subnet that corresponds to rigid component $R1$ in Figure 1(a) (inside the box with dotted borderline).

Definition 3.7 states that a task is mapped to a net transition with a single input and a single output arc. An *and* gateway maps to a transition with multiple outgoing arcs (*and* split) or multiple incoming arcs (*and* join). An *xor* gateway

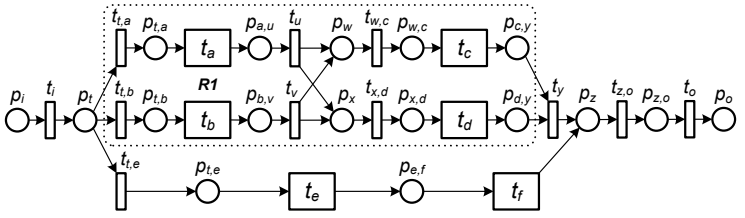


Figure 6: A WF-net that corresponds to the process model in Figure 1(a)

maps to a place with multiple outgoing arcs (*xor* split) or multiple incoming arcs (*xor* join). The places corresponding to *xor* splits are immediately followed by silent (τ) transitions representing the branching conditions, e.g., transitions $t_{t,a}$, $t_{t,b}$, and $t_{t,e}$ in Figure 6. Please note that silent transitions of a labeled net are drawn as empty rectangles.

Observe that every process model with a single source task and a single sink task always maps onto a WF-net. A process model is *sound*, if and only if its corresponding WF-net is sound. In this article, we only consider sound process models. A sound free-choice WF-system is guaranteed to be safe [33]. Hence, the rest of the article deals with sound and safe process models.

4. Behavioral Equivalence of Process Models

This section serves two purposes: (i) motivates the selection of fully concurrent bisimulation, among other notions of behavioral equivalence, for structuring of process models, (ii) discusses a procedure for checking behavioral equivalence of special nets, viz. occurrence nets, by using the notion of ordering relations.

4.1. Fully Concurrent Bisimulation

An unstructured process model and its structured version are structurally different, but behaviorally equivalent. There exist many notions of behavioral equivalence for concurrent systems [34]. A common notion is that of *bisimulation*. Related notions are those of *weak bisimulation* and *branching bisimulation*, which abstract away from silent transitions. These notions have been advocated as being suitable for comparing process models [18]. However, we argue that they are not suitable for our purposes. These three notions adopt interleaving semantics, i.e., no two tasks are executed exactly at the same time. Thus, a concurrent system and its sequential simulation are considered equivalent. For example, Figure 7 shows the sequential simulation of the net in Figure 6. The net is structured and weakly bisimilar with the net in Figure 6, but it contains no parallel branch. We could take any process model, compute its sequential simulation, structure this sequential net using GOTO program transformations, and transform the resulting sequential net into a structured process model. This structuring method is complete, but if we start with a process model containing

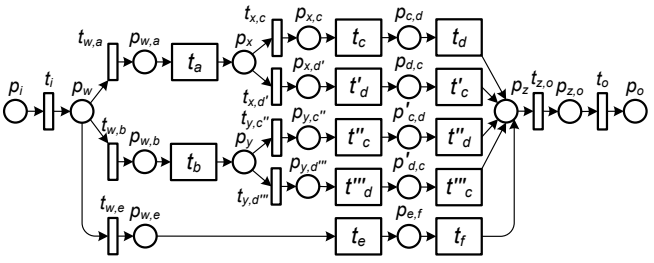


Figure 7: A sequential simulation of the net in Figure 6

and gateways, we obtain a (much larger) structured process model without any parallel branches.

Accordingly, we adopt a notion of equivalence that preserves the level of concurrency of observable transitions, viz. *fully concurrent bisimulation* [4]. Fully concurrent bisimulation is defined in terms of concurrent runs of a system, a.k.a. *processes* in the literature (but not to be confused with "business processes" or workflows). Every concurrent run of a system can be expressed as another net with a particular structure, namely a causal net.

Definition 4.1 (Causal net).
A net $N = (P, T, F)$ is a *causal* net, iff :

- for each place $p \in P$ holds $|\bullet p| \leq 1$ and $|p \bullet| \leq 1$, and
- N is acyclic, i.e., F^+ is irreflexive.

We also introduce *ordering relations* [35], which will be used throughout the article as an instrument for reasoning about the behavior of nets.

Definition 4.2 (Ordering relations).
Let $N = (P, T, F)$ be a net and let $x, y \in P \cup T$ be two nodes of N.

- x and y are in *causal* relation, written $x \rightsquigarrow_N y$, iff $(x, y) \in F^+$. We denote by $\reflectbox{$\rightsquigarrow$}_N$ the inverse of $\rightsquigarrow_N$.
- x and y are in *conflict*, written $x \#_N y$, iff there exist distinct transitions $t_1, t_2 \in T$, such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, and $(t_1, x), (t_2, y) \in F^*$. If $x \#_N x$, then x is in *self-conflict*.
- x and y are *concurrent*, written $x \|_N y$, iff neither $x \rightsquigarrow_N y$, nor $y \rightsquigarrow_N x$, nor $x \#_N y$.

The set $\mathcal{R}_N = \{\rightsquigarrow_N, \reflectbox{$\rightsquigarrow$}_N, \#_N, \|_N\}$ forms the *ordering relations* of N.

In the following, we omit the subscripts of ordering relations where the context is clear. It is easy to see that any two nodes in a causal net are either in causal relation or concurrent. In order to define a process, we lack the notion of a cut. A *cut* of a net is the maximal set of pairwise concurrent places with respect to set inclusion. Finally, a process is defined as follows.

Definition 4.3 (Process).
A *process* $\pi = (N_\pi, \rho)$ of a system $S = (N, M_0)$, $N = (P, T, F)$, consists of a causal net $N_\pi = (P_\pi, T_\pi, F_\pi)$ and a function $\rho : P_\pi \cup T_\pi \to P \cup T$, such that:

- $\rho(P_\pi) \subseteq P,\ \rho(T_\pi) \subseteq T$,
- $Min(N_\pi)$ is a cut, which corresponds to the initial marking M_0, that is $\forall\ p \in P : M_0(p) = |\rho^{-1}(p) \cap Min(N_\pi)|$, and
- $\forall\ t \in T_\pi\ \forall\ p \in P : (F(p, \rho(t)) = |\rho^{-1}(p) \cap {\bullet t}|) \wedge (F(\rho(t), p) = |\rho^{-1}(p) \cap t {\bullet}|)$.

A process π of S is *initial*, iff $T_\pi = \varnothing$.

A process π' is an *extension* of a process π if it is possible to observe π before one observes π'. Consequently, process π is a *prefix* of π'.

Definition 4.4 (Prefix, Process extension).
Let $\pi = (N_\pi, \rho)$, $N_\pi = (P_\pi, T_\pi, F_\pi)$, be a process of $S = (N, M_0)$, $N = (P, T, F)$. Let c be a cut of N_π and let $c^\downarrow$ be the set $\{x \in P_\pi \cup T_\pi \mid \exists\ y \in c : (x, y) \in F^*\}$. A process $\pi_c^\downarrow$ is a *prefix* of π, iff $\pi_c^\downarrow = ((P_\pi \cap c^\downarrow, T_\pi \cap c^\downarrow, F \cap (c^\downarrow \times c^\downarrow)), \rho|_{c^\downarrow})$. A process π' is an *extension* of process π if π is a prefix of π'.

In order to define fully concurrent bisimulation, we need two auxiliary definitions: λ-abstraction of a process, which is a process footprint that ignores silent transitions, and the order-isomorphism of λ-abstractions.

Definition 4.5 (λ-abstraction).
Let $S = (N, M_0)$, $N = (P, T, F, \mathcal{T}, \lambda)$, be a labeled system and let $\pi = (N_\pi, \rho)$, $N_\pi = (P_\pi, T_\pi, F_\pi)$, be a process of S. The *λ-abstraction* of π, denoted by $\alpha_\lambda(\pi) = (T'_\pi, \prec, \lambda')$, is defined by $T'_\pi = \{t \in T_\pi \mid \lambda(\rho(t)) \neq \tau\}$, $\prec$ is the causal relation of N_π restricted to the transitions in T'_π, i.e., $\prec\ = \rightsquigarrow_{N_\pi} \cap (T'_\pi \times T'_\pi)$, and $\lambda' : T'_\pi \to \mathcal{T}$, such that $\lambda'(t) = \lambda(\rho(t))$, $t \in T'_\pi$.

Two λ-abstractions are order-isomorphic if there exists a one-to-one correspondence between transitions of both abstractions which also preserves the ordering of the corresponding transitions in the abstractions.

Definition 4.6 (Order-isomorphism of λ-abstractions).
Let $\alpha_{\lambda_1} = (T_1, \prec_1, \lambda_1)$ and $\alpha_{\lambda_2} = (T_2, \prec_2, \lambda_2)$ be two λ-abstractions, both with labels in $\mathcal{T}$. Then α_{λ_1} and α_{λ_2} are *order-isomorphic*, iff there is a bijection $\beta : T_1 \to T_2$, such that $\forall\ t \in T_1 : \lambda_1(t) = \lambda_2(\beta(t))$ and $\forall\ t_1, t_2 \in T_1 : t_1 \prec_1 t_2 \Leftrightarrow \beta(t_1) \prec_2 \beta(t_2)$.

Given all of the above, fully concurrent bisimulation is defined as follows:

Definition 4.7 (Fully concurrent bisimulation).
Let $S_1 = (N_1, M_1)$ and $S_2 = (N_2, M_2)$ be labeled systems, $N_1 = (P_1, T_1, F_1, \mathcal{T}_1, \lambda_1)$ and $N_2 = (P_2, T_2, F_2, \mathcal{T}_2, \lambda_2)$. S_1 and S_2 are *fully concurrent bisimilar*, or *FCB-equivalent*, denoted by $S_1 \approx S_2$, iff there is a set $\mathcal{B} \subseteq \{(\pi_1, \pi_2, \beta)\}$, such that:

(i) π_1 is a process of S_1, π_2 is a process of S_2, and β is a relation between the non-τ transitions of π_1 and π_2.

(ii) If π_0^1 and π_0^2 are the initial processes of S_1 and S_2, respectively, then $(\pi_0^1, \pi_0^2, \varnothing) \in \mathcal{B}$.

(iii) If $(\pi_1, \pi_2, \beta) \in \mathcal{B}$, then β is an order-isomorphism between the λ_1-abstraction of π_1 and the λ_2-abstraction of π_2.

(iv) $\forall(\pi_1, \pi_2, \beta) \in \mathcal{B}$:
   (a) If π_1' is an extension of π_1, then $\exists\ (\pi_1', \pi_2', \beta') \in \mathcal{B}$ where π_2' is an extension of π_2 and $\beta \subseteq \beta'$.
   (b) Vice versa.

Fully concurrent bisimulation defines an equivalence relation on labeled systems that is stricter than weak bisimulation and related notions. The nets in Figure 6 and Figure 7 are not fully concurrent bisimilar. Meanwhile, the two models in Figure 1 are FCB-equivalent (with the understanding that two process models are FCB-equivalent if the corresponding Petri nets are FCB-equivalent).

4.2. Behavioral Equivalence and Ordering Relations

The above definition of fully concurrent bisimulation is abstract and hardly of any use when synthesizing structured nets from unstructured ones. Accordingly, we employ a more convenient way of reasoning about equivalence based on the ordering relations of the special class of nets, viz. *occurrence nets*.

Nets can have forward or backward conflicts, i.e., places with multiple output or input transitions, respectively. This means that a subnet which may cause a transition firing is not unique. An occurrence net is a net of a special kind. Occurrence nets forbid backward conflicts and, thus, ensure the unique cause of a transition firing. Essentially, occurrence nets generalize causal nets by allowing forward conflicts. Note that every causal net is also an occurrence net.

Definition 4.8 (Occurrence net).
A net $N = (P, T, F)$ is an *occurrence* net, iff :
- for each place $p \in P$ holds $|\bullet p| \leq 1$,
- N is acyclic, i.e., F^+ is irreflexive,
- for each node $x \in P \cup T$ the set $\{y \in P \cup T \mid (y, x) \in F^+\}$ is finite, and
- no $t \in T$ is in self-conflict, i.e., $\#_N$ is irreflexive.

Every two nodes of an occurrence net are either in causal, inverse causal, conflict, or concurrent relation [35]. Let $N = (P, T, F, \mathcal{T}, \lambda)$ be a labeled occurrence net and let $T' \subseteq T$ be its observable transitions. The λ-*ordering relations* of N are formed by its ordering relations restricted to T', i.e., $\mathcal{R}_\lambda = \{\rightsquigarrow_N \cap (T' \times T'), \reflectbox{$\rightsquigarrow$}_N \cap (T' \times T'), \#_N \cap (T' \times T'), \|_N \cap (T' \times T')\}$. We say that two ordering relations are isomorphic, if there exists a mapping between the observable transitions such that every corresponding pair of transitions is in the same ordering relation.

Definition 4.9 (Isomorphism of ordering relations).
Let $N_1 = (P_1, T_1, F_1, \mathcal{T}_1, \lambda_1)$ and $N_2 = (P_2, T_2, F_2, \mathcal{T}_2, \lambda_2)$ be two labeled occurrence nets. Let $T_1' \subseteq T_1$ and $T_2' \subseteq T_2$ be observable transitions of N_1 and N_2, respectively. Two λ-ordering relations $\mathcal{R}_{\lambda_1}$ of N_1 and $\mathcal{R}_{\lambda_2}$ of N_2 are *isomorphic*, denoted by $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$, iff there is a bijection $\gamma : T_1' \to T_2'$, such that:
- $\forall\ t \in T_1' : \lambda_1(t) = \lambda_2(\gamma(t))$, and
- $\forall\ t_1, t_2 \in T_1' : (t_1 \rightsquigarrow_{N_1} t_2 \wedge \gamma(t_1) \rightsquigarrow_{N_2} \gamma(t_2)) \vee (t_2 \rightsquigarrow_{N_1} t_1 \wedge \gamma(t_2) \rightsquigarrow_{N_2} \gamma(t_1)) \vee (t_1 \#_{N_1} t_2 \wedge \gamma(t_1) \#_{N_2} \gamma(t_2)) \vee (t_1 \|_{N_1} t_2 \wedge \gamma(t_1) \|_{N_2} \gamma(t_2))$.

Finally, we show that two occurrence nets with isomorphic ordering relations are FCB-equivalent, and vice versa. This result is exploited in the next section.

Theorem 1. *Let $S_1 = (N_1, M_1)$, $N_1 = (P_1, T_1, F_1, \mathcal{T}_1, \lambda_1)$, and $S_2 = (N_2, M_2)$, $N_2 = (P_2, T_2, F_2, \mathcal{T}_2, \lambda_2)$, be two labeled occurrence systems with natural markings and distinctive labelings. Let $T_1' \subseteq T_1$ and $T_2' \subseteq T_2$ be observable transitions of N_1 and N_2, respectively, such that there exists a bijection $\psi : T_1' \to T_2'$ for which holds $\lambda_1(t) = \lambda_2(\psi(t))$, for all $t \in T_1'$. Let $\mathcal{R}_{\lambda_1}$ and $\mathcal{R}_{\lambda_2}$ be the λ-ordering relations of N_1 and N_2, respectively. Then, it holds:*

$$S_1 \approx S_2 \;\Leftrightarrow\; \mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}.$$

Proof. We prove each direction of the equality separately.

($\Rightarrow$) Let S_1 and S_2 be FCB-equivalent. We want to show that $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$. Let us assume that $S_1 \approx S_2$ holds, but $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$ does not hold. Furthermore, let us consider transitions $t_i^1, t_j^1 \in T_1'$ that are in one-to-one correspondence with transitions $t_i^2, t_j^2 \in T_2'$, i.e., $\psi(t_i^1) = t_i^2$ and $\psi(t_j^1) = t_j^2$. All scenarios can be reduced to the following two cases:

Case 1: ($t_i^1 \parallel_{N_1} t_j^1$ or $t_i^1 \rightsquigarrow_{N_1} t_j^1$, and $t_i^2 \#_{N_2} t_j^2$). If $t_i^1 \parallel_{N_1} t_j^1$ or $t_i^1 \rightsquigarrow_{N_1} t_j^1$, then there exists process π_1 of S_1 that contains t_i^1 and t_j^1. If $t_i^2 \#_{N_2} t_j^2$, then there exists no process π_2 of S_2 that contains t_i^2 and t_j^2.

Case 2: ($t_i^1 \rightsquigarrow_{N_1} t_j^1$, and $t_j^2 \rightsquigarrow_{N_2} t_i^2$ or $t_i^2 \parallel_{N_2} t_j^2$). Let π_1 be a process of S_1 that contains t_i^1 and t_j^1, and let π_2 be a process of S_2 that contains t_i^2 and t_j^2. Then, there exists no $\phi \subseteq \psi$, such that ϕ is an order-isomorphism between λ-abstractions of π_1 and π_2.

In both cases we reach a contradiction, i.e., systems $S1$ and $S2$ cannot be FCB-equivalent if the λ-ordering relations are not isomorphic.

($\Leftarrow$) Let $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$. We want to show that S_1 and S_2 are FCB-equivalent. Let us assume that $\mathcal{R}_{\lambda_1} \cong \mathcal{R}_{\lambda_2}$ holds, but $S_1 \approx S_2$ does not hold. Then, for instance, there exists process π_1' of S_1, which has no corresponding order-isomorphic process of S_2. Suppose that π_1' has the minimal size among all such processes, i.e., any prefix of π_1' has a corresponding order-isomorphic process of S_2. Let π_1' be an extension of process π_1 of S_1 by exactly one observable transition $t_j^1 \in T_1'$. Let π_2 be a process of S_2 that is order-isomorphic with π_1. Let $t_j^2 \in T_2'$ be in one-to-one correspondence with t_j^1, i.e., $\psi(t_j^1) = t_j^2$. All scenarios can be reduced to the following three cases:

Case 1: There exists process π_2' of S_2 that contains t_j^2 and is an extension of π_2 by one observable transition. Moreover, there exists $t_i^1 \in T_1'$ in π_1, such that $t_i^1 \rightsquigarrow_{N_1} t_j^1$. However, it holds $t_i^2 \parallel_{N_2} t_j^2$, for $t_i^2 \in T_2'$, such that $\psi(t_i^1) = t_i^2$; otherwise there exists an order-isomorphism $\phi \subseteq \psi$ between π_1' and π_2'.

Case 2: There exists no process π_2' of S_2 that contains t_j^2 and is an extension of π_2. Moreover, there exists $t_i^1 \in T_1'$ in π_1, such that $t_i^1 \rightsquigarrow_{N_1} t_j^1$. However, it holds $t_i^2 \#_{N_2} t_j^2$, for $t_i^2 \in T_2'$, such that $\psi(t_i^1) = t_i^2$.

Case 3: There exists process π_2' of S_2 that contains t_j^2 and is an extension of π_2, but not by only one observable transition. Then, there exists $t_k^2 \in T_2'$, such that $t_k^2 \nsim_{N_2} t_j^2$ but π_2 does not contain t_k^2. However, $t_k^1 \in T_1'$, such that $\psi(t_k^1) = t_k^2$, is not in π_1' and, hence, $t_k^1 \nsim_{N_1} t_j^1$.

In all three cases we reach a contradiction, i.e., the λ-ordering relations cannot be isomorphic if systems S_1 and S_2 are not FCB-equivalent. □

5. Synthesis of Structured Process Models

Given an acyclic rigid process component, the main idea of the proposed structuring technique is to compute its ordering relations and to synthesize, whenever possible, a well-structured process component that exhibits the same ordering relations (recall that a well-structured process component is the one whose RPST contains only trivial, bond, and polygon components). In order to implement this idea, we encode ordering relations in a directed graph, which we call an ordering relations graph. Afterwards, we parse the graph into so called modules that show ordering relations of well-structured process components. To this end, we rely on the technique of modular decomposition [36]. With this background, below we present the following: (i) the notion of a proper complete prefix unfolding, which is essential for extracting all the needful behavioral information from rigid components, (ii) the notion of an ordering relations graph, which is a convenient abstraction of the information in the proper complete prefix unfolding, and (iii) the structuring algorithm.

5.1. Proper Complete Prefix Unfoldings

According to Theorem 1, two systems are FCB-equivalent, if they demonstrate same ordering relations. Given an (unstructured) process model, the structuring proceeds by computing ordering relations of its corresponding net. Theorem 1 operates on occurrence nets and, hence, must be adjusted to the case of the general class of systems. We accomplish the adjustment by employing the notion of complete prefix unfolding of a system [37, 38]. A complete prefix unfolding of a system is an occurrence net that explicitly represents all concurrent runs of the system. To fit the structuring use case, complete prefix unfoldings must be restricted, i.e., they must be *proper*.

A *branching process* is a formal way to capture the relation between a system and its complete prefix unfolding. The relation technically builds on a homomorphism that preserves the nature of nodes and the environment of transitions. Let $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ be two nets. A *homomorphism* from N_1 to N_2 is a mapping $h : P_1 \cup T_1 \to P_2 \cup T_2$, such that: $h(P_1) \subseteq P_2$ and $h(T_1) \subseteq T_2$, and for all $t \in T_1$, the restriction of h to $\bullet t$ is a bijection between $\bullet t$ in N_1 and $\bullet h(t)$ in N_2; correspondingly for $t\bullet$ and $h(t)\bullet$.

Definition 5.1 (Branching process).
A *branching process* of a system $S = (N, M_0)$ is a pair $\beta = (N', \nu)$, where $N' = (P, T, F)$ is an occurrence net and ν is a homomorphism from N' to N,

such that the restriction of ν to $Min(N')$ is a bijection between $Min(N')$ and M_0, and for all $p_1, p_2 \in P$ holds if $\bullet p_1 = \bullet p_2$ and $\nu(p_1) = \nu(p_2)$, then $p_1 = p_2$.

System S is referred to as the *originative* system of the branching process. Similar to the prefix relation on processes, see Definition 4.3 and Definition 4.4, a branching process can be in the prefix relation with another branching process.

Definition 5.2 (Prefix of a branching process).
Let $\beta_1 = (N_1, \nu_1)$ and $\beta_2 = (N_2, \nu_2)$ be two branching processes of a system. β_1 is a *prefix* of β_2, if N_1 is a subnet of N_2 such that: If a place belongs to N_1, then its input transition in N_2 also belongs to N_1. If a transition belongs to N_1, then its input and output places in N_2 also belong to N_1. ν_1 is the restriction of ν_2 to nodes of N_1.

The maximal branching process of a net system with respect to the prefix relation is called *unfolding* of the system. Every net system has a unique (up to isomorphism) unfolding [37].

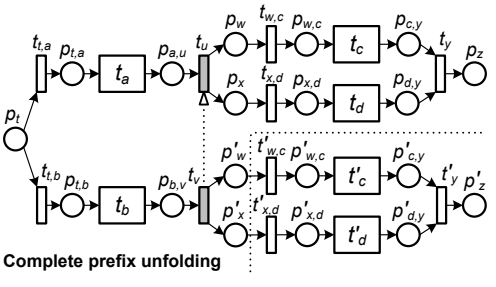


Figure 8: The unfolding and a complete prefix unfolding

Figure 8 exemplifies the unfolding of a net that corresponds to a rigid component $R1$ in Figure 6. An unfolding can contain multiple transitions that refer to the same transition in its originative system, e.g., transitions t_c and t'_c in Figure 8 both refer to transition t_c in the originative system. If we used the ordering relations computed from the unfolding to synthesize a well-structured process component, the component would contain many duplicate tasks. Fortunately, for any safe system there exists a prefix of its unfolding, called *complete prefix unfolding* [39], that is more compact than the unfolding but contains all the information about reachable markings of the originative system. Complete prefix unfoldings are finite, even for cyclic systems. A complete prefix unfolding of a system is obtained by truncating its unfolding at points where the information about reachable markings starts to be redundant. In the next definition, we summarize main notions on complete prefix unfoldings from [39].

Definition 5.3 (Complete prefix unfolding).
Let $\beta = (N', \nu)$, $N' = (P, T, F)$, be a branching process of a system $S = (N, M_0)$.

- A *configuration* C of β is a set of transitions, $C \subseteq T$, such that: (i) $t \in C$ implies that for all $t' \in T$, $t' \rightsquigarrow t$ implies $t' \in C$, i.e., C is causally closed, and (ii) for all $t_1, t_2 \in C$ holds $\neg(t_1 \# t_2)$, i.e., C is conflict-free.
- A *local* configuration of a transition $t \in T$, denoted by $\lceil t \rceil$, is the set $\{t' \in T \mid t' \rightsquigarrow t\}$, i.e., the set of transitions that precede t.
- For a finite configuration C of β, $Cut(C) = (Min(N') \cup C\bullet) \setminus \bullet C$ is a *cut*, whereas $\nu(Cut(C))$ is a reachable marking of S, denoted by $Mark(C)$.
- β is *complete* if for each reachable marking M of S there exists a configuration C of β, such that: (i) $Mark(C) = M$, i.e., M is represented in β, and (ii) for each transition t enabled at M in N, there exists a configuration $C \cup \{t'\}$, $t' \in T$, of β such that $t' \notin C$ and $\nu(t') = t$.
- A partial order $\lhd$ on the finite configurations of β is an *adequate order*, if: (i) $\lhd$ is well-founded, (ii) $C_1 \subset C_2$ implies $C_1 \lhd C_2$, and (iii) $\lhd$ is preserved by finite extensions, cf. [39] for details.
- A transition $t \in T$ is a *cutoff* transition of β, induced by $\lhd$, iff there exists a *corresponding* transition $corr(t) \in T$, such that $Mark(\lceil t \rceil) = Mark(\lceil corr(t) \rceil)$ and $\lceil corr(t) \rceil \lhd \lceil t \rceil$.
- β is a *complete prefix unfolding*, induced by $\lhd$, iff β is the maximal prefix of the unfolding of S that contains no transition after a cutoff transition.

The definition of a complete prefix unfolding is, in fact, the definition of a family of prefixes. Every adequate order leads to a different prefix with a different set of cutoff transitions. There exist several definitions of adequate orders, cf. [39–41]. Our structuring technique relies on the adequate total order for safe systems proposed in [40], hereafter denoted by $\lhd_{safe}$. The desired property of a complete prefix unfolding is minimality. For a given system, there exist no smaller complete prefix unfolding of the system than its minimal one. By employing $\lhd_{safe}$ adequate order, one always computes the minimal complete prefix unfolding of a safe system, if one only compares markings corresponding to local configurations [40].

Figure 8 exemplifies the notion of a complete prefix unfolding. The dotted lines indicate which parts of the unfolding must be truncated. Transition t_v is a cutoff transition of the unfolding, whereas transition t_u is its corresponding transition; the relation is visualized by a dotted arc. Both local configurations of transitions t_v and t_u induce the same marking $\{p_w, p_x\}$ in the originative system. The subnet of the unfolding that follows $Cut(\lceil t_v \rceil)$ is isomorphic with the subnet that follows $Cut(\lceil t_u \rceil)$ (equivalent markings imply equivalent futures of branching processes, cf. [39]). As this implies redundancy, only one subnet is included in the complete prefix unfolding.

The running time of the algorithm for constructing a complete prefix unfolding of a safe system when employing an adequate total order, cf. [39], has an upper bound of $O(|T| \cdot R^{\xi})$, where T is the set of transitions, R is the number of reachable markings, and ξ is the maximal size of the presets or postsets of the transitions in the originative system. The complete prefix unfolding contains a total number of $O(\xi \cdot R)$ nodes. Please observe that in the case of an *and* rigid component, the step of computing a complete prefix unfolding is not required, as the corresponding WF-net and its complete prefix unfolding coincide. Besides,

we do not compute a complete prefix unfolding over the whole net, but only on individual rigid components of the net. Tests we have conducted with sample process models show that the complete prefix unfolding computation takes sub-second times[2]. This finding is in line with other works that have empirically shown that complete prefix unfolding computation is efficient in practice.

In order to allow structuring, we impose an additional requirement on complete prefix unfoldings: A complete prefix unfolding must be proper.

Definition 5.4 (Proper complete prefix unfolding).
Let $\beta = (N, \nu)$, $N = (P, T, F)$, be a branching process of an acyclic system S.

- A cutoff transition $t \in T$ of β induced by an adequate order $\lhd$ is *healthy*, iff $Cut(\lceil t \rceil) \setminus t\bullet = Cut(\lceil corr(t) \rceil) \setminus corr(t)\bullet$.
- β is a *proper complete prefix unfolding*, or a *proper prefix*, induced by an adequate order $\lhd$, iff β is the maximal prefix of the unfolding of S that contains no transition after a healthy cutoff transition.

Proper prefixes of acyclic systems are clearly complete and finite. The properness requirement guarantees that concurrency is kept encapsulated, i.e., if some branch of a complete prefix unfolding contains a non-cutoff transition t that introduces concurrency, i.e., $|t\bullet| > 1$, then subsequently the very same branch must contain a transition t' that synchronizes this concurrency, i.e., for all $p \in t\bullet$ holds $p \rightsquigarrow t'$. As the restriction for healthy cutoff transitions is defined in terms of local configurations, a proper complete prefix unfolding of a safe acyclic system is always minimal when constructed using $\lhd_{safe}$ adequate order. In the following, when we refer to a proper prefix of a safe acyclic system, we always assume the minimal prefix constructed using $\lhd_{safe}$ adequate order. The only cutoff transition t_v in Figure 8 is healthy and, hence, the complete prefix unfolding is proper.

5.2. Ordering Relations Graphs

Ordering relations of a proper prefix specify the unique behavioral footprint of its originative system. Our idea is to use these relations to synthesize a well-structured process component. For this purpose, it is convenient to encode ordering relations in a directed graph, viz. *ordering relations graph*. As will be shown later, such a representation allows for a simple structuring algorithm.

In order to overcome the effects of the proper prefix truncation at healthy cutoff transitions, the notion of an ordering relations graph is founded on proper ordering relations.

Definition 5.5 (Ordering relations graph).
Let $\beta = (N, \nu)$, $N = (P, T, F)$, be a proper complete prefix unfolding of a labeled acyclic system $S = (N', M_0)$, $N' = (P', T', F', \mathcal{T}, \lambda)$.

[2]Using the Mole tool – `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`, which implements the algorithm in [39]

- Two nodes x and y of N are in *proper causal* relation, denoted by $x \nrightarrow_N y$, iff $(x, y) \in F^+$ or there exists a sequence $(t_1, \ldots, t_n)$ of healthy cutoff transitions of β, $t_i \in T$, $1 \leq i \leq n$, $n \in \mathbb{N}$, such that $(x, t_1) \in F^*$, $(corr(t_n), y) \in F^+$, and $(corr(t_j), t_{j+1}) \in F^*$, $1 \leq j < n$. We denote by $\nleftarrow_N$ the inverse of $\nrightarrow_N$.
- Let $\mathcal{R}_N = \{\rightsquigarrow_N, \nleftarrow_N, \#_N, \|_N\}$ be the ordering relations of N. The set $\boxplus_N = \#_N \setminus (\nrightarrow_N \cup \nleftarrow_N)$ is the *proper conflict* relation of N. The set $\mathcal{R}^N = \{\nrightarrow_N, \nleftarrow_N, \boxplus_N, \|_N\}$ forms the *proper ordering relations* of N. We refer to $\mathcal{R}^N$ as *observable* proper ordering relations, if the relations in $\mathcal{R}^N$ are restricted to the set of transitions of N that correspond to observable transitions of N'.
- Let $\mathcal{R}^N = \{\nrightarrow_N, \nleftarrow_N, \boxplus_N, \|_N\}$ be the observable proper ordering relations of N. An *ordering relations graph*, or *orgraph*, $\mathcal{G}_N = (V, A, \mathcal{B}, \sigma)$ of β consists of vertices $V \subseteq T$ defined by transitions of N that correspond to observable transitions of N', i.e., $V = \{t \in T \mid \lambda(\nu(t)) \neq \tau\}$, arcs $A = \nrightarrow_N \cup \boxplus_N$, and the labeling function $\sigma : V \to \mathcal{B}$, $\mathcal{B} = \mathcal{T} \setminus \{\tau\}$ with $\sigma(v) = \lambda(\nu(v))$, $v \in V$.

In the following, we omit the subscripts (superscripts) of proper ordering relations and orgraphs where the context is clear.

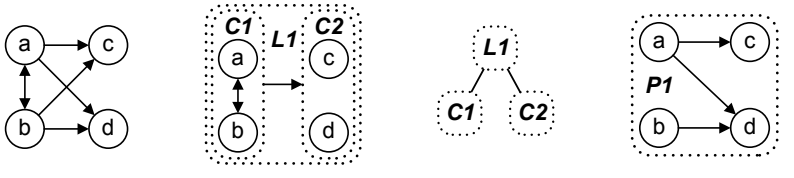


Figure 9: (a) An orgraph, (b)–(c) the MDT of (a), and (d) the MDT of the orgraph computed for the rigid component $R1$ in Figure 5

Figure 9(a) shows the orgraph of the proper complete prefix unfolding in Figure 8. Orgraphs are visualized as directed graphs. Due to a design decision, arcs of orgraphs encode proper causal (one-sided arrows) and proper conflict (two-sided arrows) relations. Such a design allows for unique encoding of proper ordering relations. Thus, one can deduce from the orgraph in Figure 9(a) that transitions t_a and t_c of the proper prefix in Figure 8 are in proper causal relation, t_a and t_b are in proper conflict relation, whereas t_c and t_d are concurrent. Observe that t_b is in proper causal relation with t_c and t_d as there exists a sequence (t_v), such that $(t_b, t_v) \in F^*$, $(corr(t_v), t_c) \in F^+$, and $(corr(t_v), t_d) \in F^+$.

5.3. Structuring

The RPST of a well-structured process model is composed of trivial, polygon, and bond (either *and* or *xor*) components. In contrast to a rigid component, each component in a well-structured model has a well-defined and regular structure within the corresponding orgraph, which allows for a precise characterization. The ordering relations graph of an *xor* bond is a complete graph, or a clique,

whereas the orgraph of an *and* bond is an edgeless graph. These topologies are consistent with the intuition behind, i.e., all tasks in an *xor* bond are in conflict, i.e., only one is executed, and all tasks in an *and* bond are concurrently executed. Figure 10(a) shows an *xor* bond with three branches, whereas Figure 10(b) shows the corresponding complete orgraph. Similarly, Figure 10(c) and Figure 10(d) show an *and* bond and the corresponding orgraph, respectively. In the cases of a trivial and polygon component, the orgraph is a direct acyclic graph representing the transitive closure, or the total order, of the proper causal relation. Figure 10(e) shows a polygon composed of three tasks, whereas Figure 10(f) presents the corresponding transitive closure over the proper causal relation.

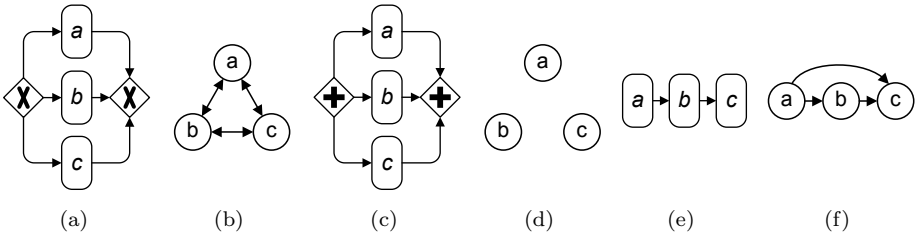


Figure 10: (a) An *xor* bond component, (b) a complete orgraph, (c) an *and* bond component, (d) an edgeless orgraph, (e) a polygon component, and (f) a total order orgraph

The main idea of our structuring technique is to parse a given orgraph into subgraphs. If one can decompose an orgraph into subgraphs such that every subgraph is either complete, edgeless, or total order, then one can construct a corresponding well-structured process component for each of the discovered subgraphs and, in this way, to synthesize a well-structured process model. To perform the parsing, we rely on the *modular decomposition* of directed graphs [36].

The modular decomposition is a technique for parsing directed graphs into modules. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph. A *module* $M \subseteq V$ of $\mathcal{G}$ is a non-empty subset of vertices of $\mathcal{G}$ that are in uniform relation with vertices $V \setminus M$, i.e., if $v \in V \setminus M$, then v has directed arcs to all members of M or to none of them, and all members of M have directed arcs to v or none of them do. However, $v_1, v_2 \in V \setminus M$, $v_1 \neq v_2$, can have different relations to members of M. Moreover, the members of M and $V \setminus M$ can have arbitrary relations to each other [36]. This definition of a module supports our intent of synthesizing a process component from the ordering relations captured in a module; all tasks inside a SESE process component are in the same ordering relation with a given task outside the component.

Two modules M_1 and M_2 of $\mathcal{G}$ *overlap*, iff they intersect and neither is a subset of the other, i.e., $M_1 \setminus M_2$, $M_1 \cap M_2$, and $M_2 \setminus M_1$ are all non-empty. M_1 is *strong*, iff there exists no module M_2 of $\mathcal{G}$, such that M_1 and M_2 overlap. The *modular decomposition* substitutes each strong module of a graph by a new vertex and proceeds recursively. The result is a unique rooted tree, called the *modular decomposition tree*, which can be computed in linear time [36].

Definition 5.6 (Modular Decomposition Tree).
The *modular decomposition tree* (MDT) of an ordering relations graph is the set of all its strong modules.

According to [36], each module in the MDT belongs to one out of four classes.

Definition 5.7 (Trivial, Linear, Complete, Primitive).
Let M be a module of an ordering relations graph $\mathcal{G}$.

- M is a *trivial* module, iff M is singleton, i.e., M contains a single vertex.
- M is a *linear* module, iff there exists a linear order $(x_1, \ldots, x_{|M|})$ of elements of M, such that there is a directed arc from x_i to x_j in $\mathcal{G}$, iff $i < j$.
- M is a *complete* module, iff the subgraph of $\mathcal{G}$ induced by vertices in M is either complete or edgeless.
- M is a *primitive* module, iff M is neither a trivial, nor a linear, nor a complete module.

For our purposes, we classify modules further: If a complete module induces a complete subgraph, the module is referred to as a *xor* complete. If a complete module induces an edgeless subgraph, the module is referred to as an *and* complete. Finally, if a module induces a subgraph with a pair of distinct vertices which are not connected by an arc, the module is said to be *concurrent*.

Considering all of the above, the next proposition summarizes relations between components of a process model and modules of an orgraph.

Proposition 5.1. Let C_1 be a process component and let M_1 be the corresponding ordering relations graph. Let M_2 be an ordering relations graph and let C_2 be its corresponding process component.

1. If C_1 is trivial or polygon, then M_1 is linear.
2. If M_2 is linear, then there exists C_2 that is trivial or polygon.
3. If C_1 is *and* (*xor*) bond, then M_1 is *and* (*xor*) complete.
4. If M_2 is *and* (*xor*) complete, then there exists C_2 that is *and* (*xor*) bond.

Figure 9(b) shows the MDT of the orgraph in Figure 9(a). Every region inside a dotted box defines a strong module. Note that module names hint at their class. For instance, module $C1$ is a complete module composed of two vertices a and b. Observe that every vertex outside $C1$, i.e., either vertex c or d, is in same relation with every vertex inside $C1$, i.e., vertices a and b. As $C1$ induces a complete graph, it is an *xor* complete module. Similarly, $C2$ is an *and* complete module. By treating both modules as singletons, the modular decomposition identifies that they are in total order and, hence, form linear module $L1$. Figure 9(c) visualizes the MDT as a tree (without trivial modules).

We are now ready to present the main result of this section.

Theorem 2. *Let $\mathcal{G}$ be an ordering relations graph. The MDT of $\mathcal{G}$ contains no primitive module, iff there exists a well-structured process model W such that $\mathcal{G}$ is the ordering relations graph of W.*

Proof. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph.

($\Rightarrow$) Let $\mathcal{G}$ be such that the MDT of $\mathcal{G}$ contains no primitive module. We show now by structural induction on the MDT of $\mathcal{G}$ that there exists a well-structured process model W such that $\mathcal{G}$ is the ordering relations graph of W. The MDT of $\mathcal{G}$ can contain trivial, linear, or complete modules.

Base: If the MDT of $\mathcal{G}$ consists of a single module M, then M is a trivial module and W is a process model composed of a single task.

Step: Let M be a module of the MDT of $\mathcal{G}$ such that every child module of M has a corresponding well-structured process component. If M is linear, then W can be a trivial or polygon component composed from children of M, cf. (2) in Prop. 5.1. If M is complete, then W can be a bond process component, either *and* or *xor*, composed from process components that correspond to child modules of M, cf. (4) in Prop. 5.1. In both cases, M has a corresponding well-structured process model (component).

Therefore, there exists a well-structured process model W that is composed of process components which correspond to child modules of module V, such that $\mathcal{G}$ is the ordering relations graph of W.

($\Leftarrow$) Let W be a well-structured process model such that $\mathcal{G}$ is the ordering relations graph of W. We show now by structural induction on the RPST of W that the MDT of $\mathcal{G}$ has no primitive module. Because W is well-structured, the RPST of W has no rigid component.

Base: If W is composed of a single task, then the corresponding ordering relations graph contains one trivial module.

Step: Let C be a process component of the RPST of W such that every child process component of C has a corresponding ordering relations graph without a primitive module. If C is trivial or polygon, then $\mathcal{G}$ is either trivial or linear, cf. (1) in Prop. 5.1. If C is bond, then $\mathcal{G}$ is complete, cf. (3) in Prop. 5.1. In both cases, C has a corresponding ordering relations graph without a primitive module.

Therefore, the MDT of the ordering relations graph that corresponds to W has no primitive module. □

Theorem 2 implicitly specifies a procedure which, given a process model, synthesis an FCB-equivalent well-structured model. This procedure is made explicit in Algorithm 1. Note that the algorithm expects as input a process model (component) in which no pair of distinct tasks have the same label.

Without loss of generality, Algorithm 1 assumes that the RPST of the process model (or process component), taken as input, consists of a single rigid component. The algorithm can be trivially extended to the case where the RPST of the process model given as input consists of a rigid component with polygons, bonds, or rigids as descendants. In this latter case, child components of the rigid component need to be abstracted into atomic task nodes. Also, if we were given a process model whose RPST contains several rigid components, we would start by structuring the rigid components at lower levels of the RPST and collapsing them into atomic task nodes before attempting to structure rigids at upper levels. The complexity of the algorithm is determined by the complexity of the unfolding step (see earlier discussion). The computation of

Algorithm 1: Structuring an Acyclic Rigid Process Component

Input: P : A process model (component) in which all tasks are distinctly labeled and whose RPST consists of an acyclic rigid component

Output: T : An RPST consisting of trivials, bonds, polygons

$N \leftarrow$ CorrespondingNet(P) // cf. Definition 3.7

$U \leftarrow$ ProperCompletePrefixUnfolding(N) // cf. Definition 5.4

$ORG \leftarrow$ OrderingRelationsGraph(U) // cf. Definition 5.5

$MDT \leftarrow$ ModularDecomposition(ORG) // cf. Definition 5.6

$T \leftarrow$ RPST obtained by traversing each module M of the MDT (in postorder) and applying the following rules:

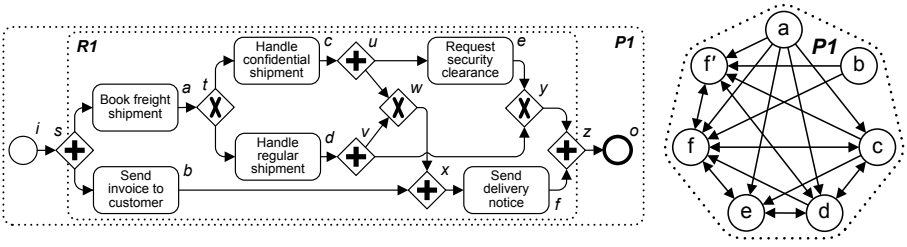
- If M is *and* complete, generate an *and* bond component in T
- If M is *xor* complete, generate an *xor* bond component in T
- If M is linear, generate a trivial or polygon component in T
- If M is non-concurrent primitive, generate a well-structured component using compiler techniques, e.g., [11]
- If M is concurrent primitive, FAIL

return T

an orgraph has polynomial complexity. All other steps can be accomplished in linear time. As explained before, in the case of an *and* rigid, the unfolding step is not required as nets which correspond to *and* rigids and their proper prefixes coincide. Note that the behavior captured in non-concurrent primitives can be structured by employing compiler techniques for GOTO program transformations. To accomplish structuring, one first needs to synthesize a program (process component) from a non-concurrent primitive module. The synthesis can be trivially accomplished by adopting the technique in [42]. The algorithm fails if the input process component is inherently unstructured, such as component $R1$ in Figure 11(a). In this particular case, the ordering relations graph of $R1$ forms a single concurrent primitive module, cf. Figure 11(b). Observe that the unfolding step duplicates the transition which corresponds to task f. As structuring relies on minimal proper prefixes (see the discussion in Section 5.1), the proposed technique always operates with the minimum duplication of transitions which is required to allow structuring.

In Figure 9(d), we show the MDT of the orgraph computed for process component $R1$ in Figure 5. The MDT contains a single primitive module $P1$, which hints at inherent unstructured nature of $R1$. Note that due to the characteristic topology of causal relations, the pattern in Figure 5 is often referred to as Z-structure or N-Structure.

In light of the above, we conclude that given an acyclic process model with an arbitrary topology, one can construct an FCB-equivalent well-structured process model, iff the proper complete prefix unfolding of the system that corresponds to the given process model is such that the modular decomposition of its orgraph contains no concurrent primitive module.



(a) (b)

Figure 11: (a) An unstructured process model with inherently unstructured rigid process component $R1$ and (b) the ordering relations graph of $R1$ from (a)

6. Structuring Multi-Source and/or Multi-Sink Process Models

In Section 5, we assumed that process models have single source and single sink nodes. However, contemporary notations for business process modeling, including BPMN and EPC, support the definition of models with multiple source and/or multiple sink events, hereafter called multi-source and multi-sink models. In this section, we extend the notion of structuredness for multi-source and multi-sink models and we extend the structuring method accordingly.

6.1. Notion of Structuredness

In the context of single-source and single-sink process models, we have defined a well-structured process model as one whose RPST does not contain any rigid components, cf. Section 2.2. This notion needs to be extended for the case of multi-source and multi-sink models. To this end, a process model with multiple source nodes is augmented with an additional (start) node and an arc from this fresh node to each of the source nodes. This additional node is labeled s by convention. Conversely, a model with multiple sink nodes is augmented with an additional (end) node and an arc from each of the sink nodes of the original model to this fresh node (labeled e by convention). This augmentation is captured by the following definition.

Definition 6.1 (Augmented process model).
Let W be a multi-source and multi-sink process model. The *augmented* version of W is constructed from W as follows:

- If W has more than one source, a new source *start* is added and for each source node s of W, an arc from *start* to s is added.
- If W has more than one sink, a new sink *end* is added and for each sink node s of W, an arc from s to *end* is added.

A multi-source and multi-sink model is said to be (well-)structured, iff the RPST of its augmented version contains no rigid process component. Please note that the augmentation of a model aligns well with the principles of the generalized RPST computation, cf. [21] for details.

Given an unstructured multi-source and multi-sink process model P, our goal is to compute an FCB-equivalent structured process model P'. By definition, this means that the labels of the nodes in P' must coincide with those in P. Hence, the special nodes s and e, which are added for the sake of constructing the RPST, need to be removed at the end of the structuring procedure, thereby yielding a structured multi-source and multi-sink process model as output.

6.2. Instantiation Semantics

Given a multi-source and multi-sink process model, we can compute the RPST of the augmented model in order to separate rigid components from non-rigid ones. Non-rigid components are already structured and thus can be replaced with a single "black box", so that their parent node in the RPST can be structured. For example, Figure 12(a) shows a multi-source and multi-sink model, captured in EPC notation. Figure 12(b) shows its augmented version whose RPST contains rigid component $R1$, under which we can find two bond components $B1$ and $B2$. Each of these bond components in turn contains two polygons, but the polygons are not shown in the figure for the sake of simplicity. After replacing components $B1$ and $B2$ with black-boxes, we obtain the abstract model depicted in Figure 12(c) consisting of a single rigid component. The problem of structuring the original EPC is then reduced to that of structuring this rigid component.

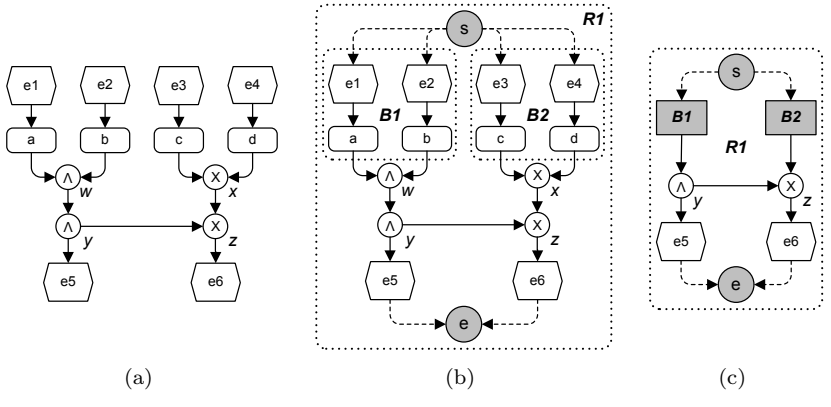


Figure 12: (a) A multi-source and multi-sink EPC, (b) its augmented version, and (c) its abstract version

Rigid components can be classified into those that contain one of the special nodes s or e, and those that do not. The latter type of rigid can be structured using the method outlined in the previous sections. Therefore, we can focus our attention on the case where a rigid contains s or e. Without loss of generality, we assume below that we are given a rigid component that contains both the s node and the e node introduced during the augmentation. The case where a rigid contains the s node or the e node (but not both) is just a special case.

The first step in structuring a rigid component is to compute its corresponding net. Definition 3.7 can be directly applied to multi-sink process models (and thus to multi-sink rigid components). The resulting net will have multiple sink places, but this feature does not pose any particular problem. Indeed, the unfolding is defined for multi-sink nets in the same way as for single-sink nets. On the other hand, the mapping of multi-source process models to Petri nets requires special care for two reasons:

1. In order to compute an unfolding, we need to define an "initial marking". In the case of a single-source net, the initial marking is the one that contains a single token in the source place and no other tokens, but in the case of multi-source nets, several initial markings are possible.
2. Different process modeling notations adopt different semantics for multi-source models [43].

Hence, we need to consider each process modeling notation separately in order to determine how to map a multi-source process model (or a process component) in that notation into a Petri net, and how to determine the initial marking from which the unfolding will be computed. Below we address these questions in the context of two concrete process modeling notations, namely BPMN and EPC.

6.2.1. Multi-source BPMN Models

The notion of process model (as per Definition 2.1) allows us to generically represent models in several graph-oriented process modeling notations, including BPMN and EPC. Below, we consider the case where a process model represents a BPMN model. In this context, a node in a process model represents either a BPMN activity, a BPMN event, or a BPMN gateway. Such process models may contain multiple source nodes, each one corresponding either to a "start event" or to an "event-based gateway". As per the BPMN standard specification [1], the instantiation semantics of such multi-source models is as follows:

- If a BPMN model starts with multiple events, a new process instance is created whenever one of these "start events" fires. The mapping of this case to a Petri net with a single start place is depicted in Figure 13(a).
- If a BPMN model starts with multiple event-based gateways that participate in a common conversation, each of these gateways must receive one token. The corresponding Petri net mapping is shown in Figure 13(b).
- If a BPMN model starts with a so-called "parallel event-based gateway", then each one of the events connected to this parallel event-driven gateway must occur before the process is instantiated. The corresponding Petri net mapping is shown in Figure 13(c).

In all three cases, we see that a multi-source BPMN model – and consequently a rigid component in a BPMN model containing the s node – can be mapped to a Petri net with a single source place, such that the initial marking consists of exactly one token in this source place. Since this mapping is trivial, as shown in Figure 13, we omit its formal definition. The rest of the section focuses on the EPC semantics, which requires a more extensive treatment.

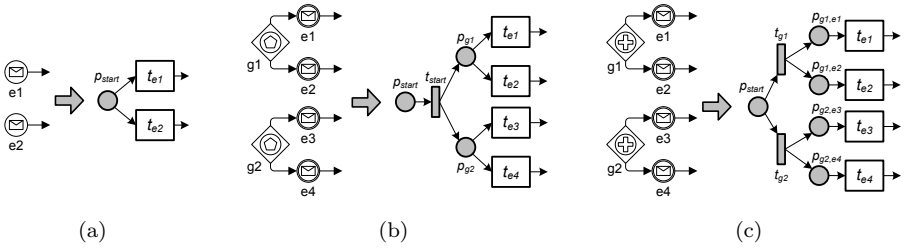


Figure 13: Mapping multi-source BPMN models to nets

6.2.2. Multi-source EPC Models

There is no "official" precise instantiation semantics for EPCs with multiple start events. However, some authors have adopted the following semantics [43]:

- An instance of the process requires at least one of the start events to be triggered.
- Additional start events may be triggered during the execution of a process instance.

An EPC can be represented as a process model (as per Definition 2.1) where each node of the process model represents either a function, an event, or a connector. In order to capture the instantiation semantics of an EPC process model with multiple source tasks, the nets obtained by applying Definition 3.7 can be augmented to a single source.

Definition 6.2 (Augmented Petri net).
Let $N = (P, T, F)$ be a net and let $S \subset P$ be the set of all source places of N. The *augmented* version of N is constructed from N as follows:

- A fresh place p_{start} is added in the net.
- For every non-empty subset of source places $\mathbf{s} \in \mathcal{P}(S) \setminus \varnothing$, a fresh *start* transition $t_{\mathbf{s}}$ and a fresh flow arc $(p_{start}, t_{\mathbf{s}})$ is added in the net.
- For every source place $s \in S$ and for every non-empty subset of source places $\mathbf{s} \in \mathcal{P}(S) \setminus \varnothing$ containing s, i.e., $s \in \mathbf{s}$, a fresh flow arc $(t_{\mathbf{s}}, s)$ is added in the net.

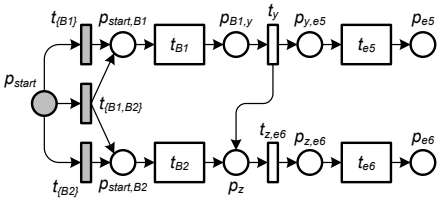


Figure 14: The augmented version of the net that corresponds to the abstract model in Figure 12(c)

For example, Figure 14 shows the augmented version of the net which corresponds to the abstract EPC model in Figure 12(c) (note the abuse of notation for

simplicity reasons). Fresh nodes are highlighted with grey background. Place p_{start} is the only source place of the resulting net.

If we set the initial marking to be the one that contains a single token in the source place (and no tokens elsewhere), the augmented net captures all possible *instantiations* of the process model. The Petri net of a multi-source and multi-sink EPC model (or component of a model) starts with one transition per possible instantiation. For instance, the net in Figure 14 starts with three transitions: transition $t_{\{B1\}}$ corresponds to the instantiation where only component $B1$ is executed, transition $t_{\{B2\}}$ corresponds to the instantiation where only component $B2$ is executed, and transition $t_{\{B1,B2\}}$ corresponds to the case where both components are executed.

6.3. Soundness

An instantiation of a process model does not always lead to a successful completion of the process. Some instantiations may lead to *deadlocks* or *lack of synchronization*. Here, a deadlock is defined as a situation where no transition can fire, but one of the branches is still active. In other words, a deadlock occurs when there is a token in a non-sink place in the net, and no transition in the net is enabled. Lack of synchronization refers to the situation in which a transition can fire twice (without any other transition firing in-between these two firings). This corresponds to the situation where the net reaches a marking where there is more than one token in a place. Deadlock freeness and proper synchronization, i.e., absence of any state exhibiting a lack of synchronization, correspond to the notions of soundness and safeness introduced earlier in this article [24].

In light of the above, Decker and Mendling [43] suggest – but do not formally define – a notion of "correct instantiation" of an EPC based on the idea that a start event will be triggered, if and only if it is required, meaning that:

- If the execution of a process instance runs into a deadlock because one of its join gateways is waiting for one of its branches to complete, and the completion of this branch requires one of the start events to be triggered, this start event will eventually be triggered so that the execution of the process instance can complete.
- A start event will not be triggered if this may eventually cause a *lack of synchronization*.

In order to illustrate the notions of deadlock and lack of synchronization, let us go back to the EPC in Figure 12(a). The execution of the process may start with an occurrence of event $e1$. Eventually, the left-hand side branch of gateway w completes, i.e., a token reaches the left-hand side incoming arc of the gateway. This is the state depicted in Figure 15(a). In order to proceed, a token is required in the second incoming branch. Otherwise, the execution would remain deadlocked in gateway w. Thus, event $e2$ must eventually occur. On the other hand, we note that neither event $e3$ nor event $e4$ may occur at this stage, since that would lead to a lack of synchronization, depicted in Figure 15(b), which shows a state where event $e6$ may occur twice.

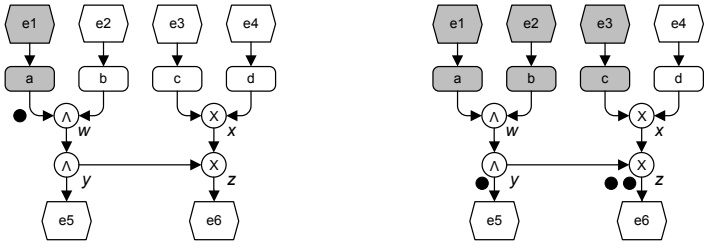


Figure 15: Markings of EPCs with (a) "event must occur" situation, and (b) lack of synchronization

Coming back to the abstract model in Figure 12(c), we observe that the instantiation where either $B1$ or $B2$ are executed is correct. On the other hand, the instantiation where both $B1$ and $B2$ are executed leads to the lack of synchronization depicted in Figure 15(b). In other words, start transition $t_{\{B1,B2\}}$ in Figure 14 corresponds to an incorrect instantiation of the original model. We note in passing that in the abstract EPC, the deadlock situation depicted in Figure 15(a) does not manifest itself because the underlying events have been abstracted away inside $B1$.

In order to capture the notion of "correct instantiation" introduced by Decker and Mendling [43], we proceed as follows: We start with the augmented version of a net that corresponds to a multi-source process model, as per Definition 6.2. Based on this net, we compute an unfolding starting from the initial marking that puts one token in the source place and no tokens elsewhere. This unfolding captures both the correct and incorrect instantiations. Accordingly, we "prune" the unfolding in order to remove those branches that represent incorrect instantiations. To this end, we formally capture – at the level of the unfolding – the notion of incorrect instantiation, i.e., lack of synchronization and deadlock. The following definition – based on similar definitions by Fahland [44] – captures these notions. Here, the term "locally unsafe place" is used in lieu of "lack of synchronization".

Definition 6.3 (Local safeness, Local deadlock).
Let $\beta = (N, \nu)$, $N = (P, T, F)$, be the unfolding of an acyclic system $S = (N', M_0)$.

- A place $p \in P$ is *locally safe* in β, iff there exists no place $q \in P$, $q \neq p$, such that p is concurrent to q and both correspond to the same place in N', i.e., $\nexists\ q \in P, q \neq p : (p \parallel_N q) \wedge (\nu(p) = \nu(q))$; otherwise p is *locally unsafe*.
- A place $p \in P$ is a *local deadlock* in β, iff one of the following holds:
  - $p\bullet = \varnothing$ and $\nu(p)\bullet \neq \varnothing$.
  - There exist transition $t \in p\bullet$, place $p' \in \bullet t$, and place $p'' \in P$, such that: $p''\bullet = \varnothing$, $p''\ \#_N\ p$, and $p''\ \|_N\ p'$.

A locally unsafe place in a branching process of a net system clearly signals that the system is unsafe, cf. Prop. 6.3 in [39]. Every local deadlock hints at existence

of a deadlock in the system. Definition 6.3 specifies the deadlock condition in the special case of acyclic systems. In case of the general class of systems, one can deduce deadlock conditions by following the principles described in [45, 46]. Accordingly, the *sound* unfolding of a system is a prefix of its unfolding that excludes incorrect instantiations.

Definition 6.4 (Sound unfolding).
Let $\beta = (N, \nu)$, $N = (P, T, F)$, be the unfolding of an acyclic system S. The *sound unfolding* of S is the maximal prefix of β that contains no transition that is in causal relation either with a locally unsafe place or a local deadlock in β.

Figure 16 exemplifies the sound unfolding of the net in Figure 14. The subnet of the unfolding below the dashed line must be pruned out because it contains a lack of synchronization. Indeed, places p''_z, $p''_{z,e6}$, p''_{e6}, p'''_z, $p'''_{z,e6}$, and p'''_{e6} (highlighted with grey background) are locally unsafe. Accordingly, transition $t_{\{B1,B2\}}$ (highlighted with black background) is the transition that is causal with all locally unsafe places of the unfolding.

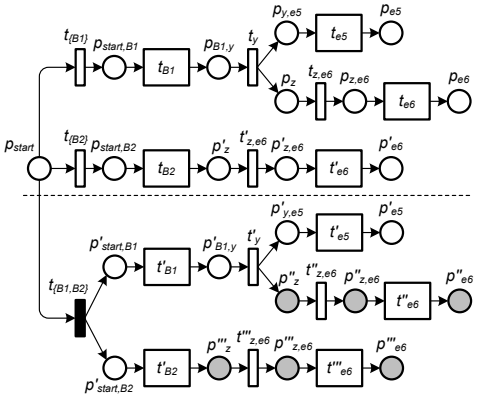


Figure 16: The sound unfolding of the net in Figure 14

In certain cases, some transitions of a system might be not represented in its sound unfolding. This happens when a transition does not occur in any execution that starts with a correct instantiation. Such systems are unsound and are excluded from further consideration.

Definition 6.5 (Soundness of acyclic nets).
Let $\beta = (N, \nu)$, $N = (P, T, F)$, be the sound unfolding of the augmented version $N' = (P', T', F')$ of an acyclic net N''. Let $S' \subseteq T'$ be the set of all start transitions of N'. N'' is said to be *sound*, iff for every $t' \in T' \setminus S'$ there exists $t \in T$, such that $\nu(t) = t'$.

Specifically, we say that a multi-source and multi-sink process model is sound, if every non-start transition of the augmented version of the corresponding

net appears in at least one execution that starts with a correct instantiation. The sound unfolding in Figure 16 represents all the non-start transitions of its originative net in Figure 14 and, hence, the model in Figure 12 is sound.

6.4. Structuring

Given a sound multi-source and multi-sink process model, one can construct an equivalent structured model using Algorithm 1 with the following changes:

- The corresponding net must be augmented according to Definition 6.2.
- One must construct a proper prefix based on the sound unfolding, i.e., as a prefix of the sound unfolding.

Note that the sound unfolding of the net in Figure 14 and its proper prefix coincide, see in Figure 16.

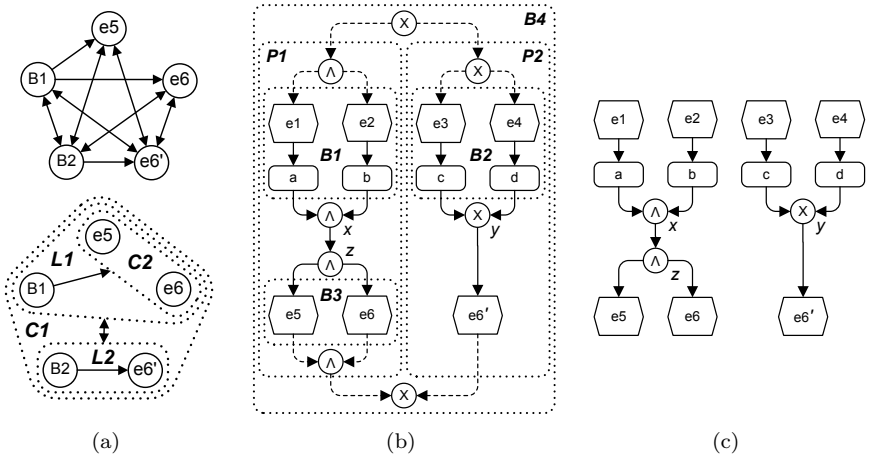


Figure 17: Structuring of the EPC in Figure 12(a): (a) the orgraph and its MDT, (b)–(c) structured versions of the EPC

Figure 17(a) shows the orgraph of the proper prefix in Figure 16, along with its MDT. The MDT contains no concurrent primitive and, therefore, the well-structured process model exhibiting given ordering relations exists. The resulting structured model has a single source and single sink, and starts and ends with gateways that encode instantiations and completions of the model, as shown in Figure 17(b), which is the well-structured version of the EPC in Figure 12(a). The figure also visualizes the process components: $B1$ and $B2$ are the components from Figure 12(b). Polygons $P1$ and $P2$ are synthesized from modules $L1$ and $L2$, respectively, cf. the MDT. Finally, bonds $B3$ and $B4$ correspond to modules $C1$ and $C2$, respectively. The gateways at the start and at the end can be trivially removed through a post-processing step, thereby yielding a structured multi-source and multi-sink EPC model, cf. Figure 17(c).

7. Evaluation

The proposed structuring method has been implemented as a tool, namely `bpstruct`, publicly available at `https://code.google.com/p/bpstruct/`. Using this implementation, we conducted an empirical evaluation of the proposed method with the aim of addressing the following questions in the context of a repository of process models taken from commercial practice:

- Q1. What proportion of unstructured process models are inherently unstructured and what proportion of models are structurable?
- Q2. Is the exponential worst-case complexity of the structuring method (particularly the unfolding method) problematic in practice?
- Q3. In theory, the structuring method may lead to node duplication. To what extent does this duplication lead to larger process models?
- Q4. The method for structuring multi-source models may lead to disconnected models. How often does this phenomenon occur?

In the following, we present the dataset which we used for the evaluation (Section 7.1), and discuss answers to the questions proposed above (Section 7.2), which we derived from the evaluation.

7.1. Dataset

For the evaluation, we used the SAP Reference Model [47] – a collection consisting of 604 EPCs capturing business processes supported by the SAP R/3 enterprise system. In the first stage, we discarded models that are already structured and models that contain cycles. We also discarded models that contained *or* joins in a rigid, since these models cannot be processed by the proposed method. Note that *or* joins at the boundaries of bonds do not pose any problem to `bpstruct`, since bonds are separated from rigids during the computation of the RPST, and `bpstruct` only needs to deal with rigids; a description of the execution semantics of *or* gateways can be found in [48]. After this pre-processing, we were left with 78 models, 40 of which are sound. As many models in the SAP Reference Model are multi-source and multi-sink, we checked soundness using the technique proposed in Section 6.3. Coincidently, each of the 40 sound models contained exactly one rigid component. Thus, the number of rigids that needed to be structured was also 40.

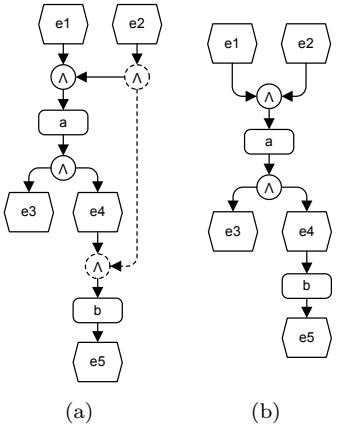


Figure 18: Structuring of a homogeneous *and* rigid

Among these 40 rigids, 6 are homogeneous *xor* rigids, 19 are homogeneous *and* rigids, and 15 are heterogeneous rigids. All homogeneous *xor* rigids can be structured, since the only source of inherent unstructuredness in acyclic process models stems from concurrency. On the other hand, all but one of the

19 homogeneous *and* rigids are inherently unstructured. The only *and* rigid that is structurable is a case of a rigid that contained redundant transitive arcs; the core structure of the rigid is summarized in Figure 18(a). The arc going from the *and* split after event *e2* to the *and* join after event *e4* is redundant and, thus, can be removed; in doing so, the split and the join become redundant and can also be removed. The structured version of the EPC in Figure 18(a) is proposed in Figure 18(b). All other 18 homogeneous *and* rigids contain the structure depicted in Figure 5. Finally, among the 15 heterogeneous rigids, only 3 are inherently unstructured. The complete characterization of the dataset employed for the evaluation is depicted in Figure 19. From the total of 604 models, we did not address 31 EPCs with cycles and 96 EPCs with *or* gateways; these are depicted by empty circles in the figure. We plan to address these cases as a part of our future work.

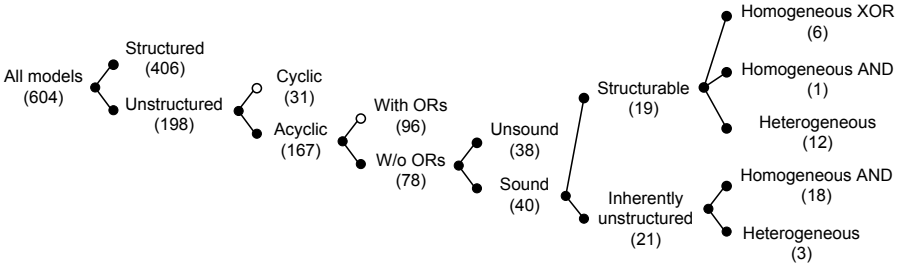
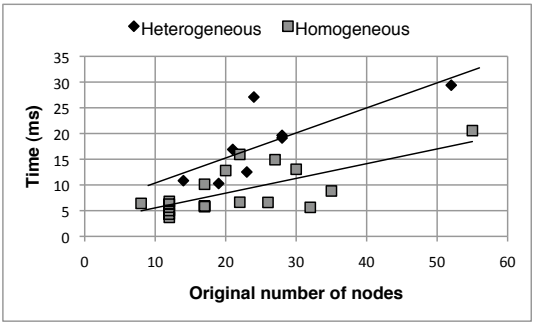


Figure 19: Structural characterization of models in the SAP Reference Model

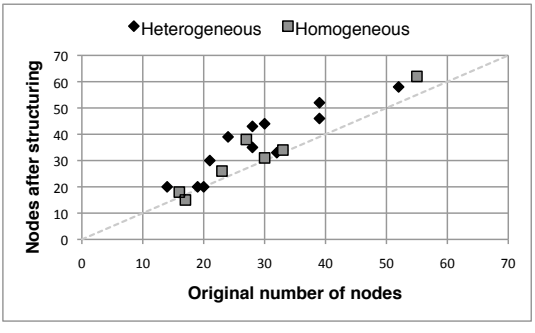
7.2. Results

With reference to question Q1 above, the evaluation suggests that unstructured homogeneous *and* rigids are highly prone to being inherently unstructured, while heterogeneous rigids are prone to be structurable. With reference to question Q2 above, Figure 20(a) plots the execution time relative to the size of the input model. The figure also plots two trendlines of the linear regression analysis: one for the heterogeneous and one for the homogeneous case. The plot shows that, although in theory the complexity of the structuring algorithm is exponential to the size of the input (due to the unfolding step – cf. Section 5), this worst-case exponential complexity does not manifest itself in the dataset at hand. The observed execution times are rather linear relative to the size of the input models. In the given dataset, we found one rigid component that contained 2 *xor* gateways intermingled with 8 *and* gateways. The structuring algorithm took 1.5 seconds to execute for this model and concluded that the model is inherently unstructured. These 1.5 seconds were spent mainly on the unfolding and the pruning steps. Putting this case aside, the average execution time for the remaining models is 12 ms (standard deviation: 7ms). These execution times exclude the time to compute the RPST, but this step has a linear complexity.

With reference to question Q3, Figure 20(b) plots the size of the structured process models relative to the size of the original models (only for models that



(a) Average structuring time



(b) Duplication ratio

Figure 20: Experimental results

were originally unstructured but not inherently unstructured). The figure shows that – except for the model with a homogeneous *and* rigid (located slightly below the diagonal), the size of the structured model is at least equal and in most cases larger than that of the input model. Specifically, we observed that the size of the output model (measured in terms of number of nodes) was up to 1.625 times that of the input models. On average, the size of the models produced by `bpstruct` was 1.22 times the size of the input model (standard deviation: 0.2). These results emphasize the fact that structuring a process model involves a tradeoff between modularity and size. Please note that duplication results were obtained for flat models, i.e., models without subprocesses. Structured models may also contain duplicates of whole process components, which leaves opportunities for modularization, e.g., by employing techniques like [29].

Finally, with reference to question Q4, we found that 15 of the 19 structurable models led to disconnected structured models. Two of the rigids whose structured versions were connected originally had a single source, while two of them were multi-source. And as expected, all rigids whose structured versions are disconnected were originally multi-source models. This result suggests that when structuring multi-source models, one is likely to obtain disconnected models. This phenomenon is specific to EPCs. It does not occur in the case of multi-source

BPMN models, since in BPMN, the multiple disconnected fragments would be re-connected with a common event-driven or parallel event-driven gateway.

8. Conclusion

We conclude that a sound acyclic process model is inherently unstructured, if and only if its RPST contains a rigid process component for which the modular decomposition of its orgraph contains a concurrent primitive. In all other cases, Algorithm 1 applied to every rigid process component of a process model constructs an FCB-equivalent well-structured process model. We have thus provided a characterization of the class of well-structured acyclic process models under fully concurrent bisimulation, and a complete structuring method, which has been implemented in the `bpstruct` tool.

Our method can also be used to structure models with SESE cycles, even if these cycles contain unstructured components. In this case, the unstructured components and the cycles are in different nodes of the RPST. However, the proposed method cannot deal with models with arbitrary cycles. We believe that the notion of the proper complete prefix unfolding, cf. Definition 5.4, can be applied to unfold all cycles in the model to the point where structuring can be addressed as a combination of the compiler techniques for structuring sequential programs and the approach proposed in this article. We plan to investigate this option in future work. Also, the proposed method does not apply to models with *or* joins and complex gateways, except in the case where these gateways appear at the boundaries of a bond. When *or* joins are present inside a rigid, the structuring problem becomes conceptually similar to the problem of transforming a process model with *or* joins into FCB-equivalent process model with *xor* and *and* gateways only, so that the unfolding techniques can be applied. Dealing with complex gateways would require different techniques since complex gateways lead to unsafe models, i.e., ones that have more than one token in the same place. Future work will aim at addressing these and other restrictions of the presented technique, such as the ability to deal with BPMN models with attached events (both interrupting and non-interrupting ones).

As mentioned in Section 1, one of the motivating reasons for structuring process models is to be able to apply existing analysis techniques which only work for structured process models. For example, existing techniques for calculating Quality of Service (QoS) properties of business processes [8] are only applicable for structured process models. In a separate work [49], we have shown how these techniques can be extended to models with arbitrary topology by applying `bpstruct` in conjunction with additional post-processing steps to deal with inherently unstructured rigid components.

Another potential benefit of structuring a process model is that the resulting structured model may be easier to comprehend and less error-prone to maintain thanks to its higher degree of modularity [5]. However, the empirical evaluation reported in this article has put into evidence that the structured process models produced by `bpstruct` are often larger than the original ones (by an average of 22% in the case of the SAP Reference Model). Larger models are generally

more difficult to comprehend and maintain than smaller models. An interesting direction for future work is to conduct empirical evaluations with end-users in order to determine whether the complexity reduction due to the modularity of the structured model outweighs the increase in complexity due to the larger size of the structured model.

Acknowledgments. This research is partly funded by ERDF via the Estonian Center of Excellence in Computer Science and the Estonian Science Foundation.

References

[1] Object Management Group (OMG), Business Process Model and Notation (BPMN) Version 2.0.
URL `http://www.omg.org/spec/BPMN/2.0/PDF/`

[2] A.-W. Scheer, O. Thomas, O. Adam, Process modeling using event-driven process chains, in: Process-Aware Information Systems, Wiley InterScience, 2005, Ch. 6, pp. 119–145.

[3] B. Kiepuszewski, A. H. M. ter Hofstede, C. Bussler, On structured workflow modelling, in: Conference on Advanced Information Systems Engineering (CAiSE), Vol. 1789 of Lecture Notes in Computer Science, Springer, 2000, pp. 431–445.

[4] E. Best, R. R. Devillers, A. Kiehn, L. Pomello, Concurrent bisimulations in Petri nets, Acta Informatica (ACTA) 28 (3) (1991) 231–264.

[5] R. Laue, J. Mendling, The impact of structuredness on error probability of process models, in: Information Systems Technology and its Applications (UNISCON), Vol. 5 of Lecture Notes in Business Information Processing, Springer, 2008, pp. 585–590.

[6] H. D. Rombach, A controlled experiment on the impact of software structure on maintainability, IEEE Transactions on Software Engineering (TSE) 13 (3) (1987) 344–354.

[7] V. R. Gibson, J. A. Senn, System structure and software maintenance performance, Communications of the ACM (CACM) 32 (3) (1989) 347–358.

[8] M. Laguna, J. Marklund, Business Process Modeling, Simulation, and Design, Prentice Hall, 2005.

[9] C. Combi, R. Posenato, Controllability in temporal conceptual workflow schemata, in: Business Process Management (BPM), Vol. 5701 of Lecture Notes in Computer Science, Springer, 2009, pp. 64–79.

[10] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, W. M. P. van der Aalst, From BPMN process models to BPEL web services, in: International/European Conference on Web Services (ICWS), IEEE Computer Society, 2006, pp. 285–292.

[11] G. Oulsnam, Unravelling unstructured programs, The Computer Journal (CJ) 25 (3) (1982) 379–387.

[12] R. Liu, A. Kumar, An analysis and taxonomy of unstructured workflows, in: Business Process Management (BPM), Vol. 3649 of Lecture Notes in Computer Science, Springer, 2005, pp. 268–284.

[13] R. Hauser, M. Friess, J. M. Küster, J. Vanhatalo, An incremental approach to the analysis and transformation of workflows using region trees, IEEE Transactions on Systems, Man, and Cybernetics, Part C (TSMC) 38 (3) (2008) 347–359.

[14] A. Polyvyanyy, L. García-Bañuelos, M. Weske, Unveiling hidden unstructured regions in process models, in: OTM Conferences, Vol. 5870 of Lecture Notes in Computer Science, Springer, 2009, pp. 340–356.

[15] R. Hauser, J. Koehler, Compiling process graphs into executable code, in: Generative Programming and Component Engineering (GPCE), Vol. 3286 of Lecture Notes in Computer Science, Springer, 2004, pp. 317–336.

[16] J. Koehler, R. Hauser, Untangling unstructured cyclic flows - a solution based on continuations, in: CoopIS/DOA/ODBASE, Vol. 3290 of Lecture Notes in Computer Science, Springer, 2004, pp. 121–138.

[17] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, J. Mendling, From business process models to process-oriented software systems, ACM Transactions on Software Engineering and Methodology (TOSEM) 19 (1).

[18] B. Kiepuszewski, A. H. M. ter Hofstede, W. M. P. van der Aalst, Fundamentals of control flow in workflows, Acta Informatica (ACTA) 39 (3) (2003) 143–209.

[19] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[20] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, in: Business Process Management (BPM), Vol. 6336 of Lecture Notes in Computer Science, Springer, 2010, pp. 276–293.

[21] A. Polyvyanyy, J. Vanhatalo, H. Völzer, Simplified computation and generalization of the refined process structure tree, in: Web Services and Formal Methods (WS-FM), Vol. 6551 of Lecture Notes in Computer Science, Springer, 2010, pp. 25–41.

[22] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, in: Business Process Management (BPM), Vol. 5240 of Lecture Notes in Computer Science, Springer, 2008, pp. 100–115.

[23] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, Data & Knowledge Engineering (DKE) 68 (9) (2009) 793–818.

[24] J. Vanhatalo, H. Völzer, F. Leymann, Faster and more focused control-flow analysis for business process models through SESE decomposition, in: International Conference on Service Oriented Computing (ICSOC), Vol. 4749 of Lecture Notes in Computer Science, Springer, 2007, pp. 43–55.

[25] F. Zhang, E. H. D'Hollander, Using hammock graphs to structure programs, IEEE Transactions on Software Engineering (TSE) 30 (4) (2004) 231–245.

[26] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, F. Cao, Compiling business processes: Untangling unstructured loops in irreducible flow graphs, International Journal of Web and Grid Services (IJWGS) 2 (1) (2006) 68–91.

[27] B. Weber, M. Reichert, J. Mendling, H. A. Reijers, Refactoring large process model repositories, Computers in Industry (CII) 62 (5) (2011) 467–486.

[28] R. Dijkman, B. Gfeller, J. Küster, H. Völzer, Identifying refactoring opportunities in process model repositories, Information and Software Technology 53 (9) (2011) 937–948.

[29] R. Uba, M. Dumas, L. García-Bañuelos, M. L. Rosa, Clone detection in repositories of business process models, in: Business Process Management (BPM), Lecture Notes in Computer Science, Springer, 2011, pp. 248–264.

[30] W. M. P. van der Aalst, Verification of workflow nets, in: Applications and Theory of Petri Nets (ICATPN/APN), Vol. 1248 of Lecture Notes in Computer Science, Springer, 1997, pp. 407–426.

[31] E. Best, Structure theory of Petri nets: the free choice hiatus, in: Advances in Petri Nets, Vol. 254 of Lecture Notes in Computer Science, Springer, 1987, pp. 168–205.

[32] J. Desel, J. Esparza, Free Choice Petri Nets (Cambridge Tracts in Theoretical Computer Science), Cambridge University Press, 1995.

[33] W. M. P. van der Aalst, Workflow verification: Finding control-flow errors using Petri-net-based techniques, in: Business Process Management (BPM), Vol. 1806 of Lecture Notes in Computer Science, Springer, 2000, pp. 161–183.

[34] R. J. van Glabbeek, The linear time-branching time spectrum (extended abstract), in: International Conference on Concurrency Theory (CONCUR), Vol. 458 of Lecture Notes in Computer Science, Springer, 1990, pp. 278–297.

[35] M. Nielsen, G. D. Plotkin, G. Winskel, Petri nets, event structures and domains, Part I, Theoretical Computer Science (TCS) 13 (1981) 85–108.

[36] R. M. McConnell, F. de Montgolfier, Linear-time modular decomposition of directed graphs, Discrete Applied Mathematics (DAM) 145 (2) (2005) 198–209.

[37] J. Engelfriet, Branching processes of Petri nets, Acta Informatica (ACTA) 28 (6) (1991) 575–591.

[38] J. Esparza, K. Heljanko, Unfoldings – A Partial-Order Approach to Model Checking, EATCS Monographs in Theoretical Computer Science, Springer, 2008.

[39] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan's unfolding algorithm, Formal Methods in System Design (FMSD) 20 (3) (2002) 285–310.

[40] J. Esparza, S. Römer, W. Vogler, An improvement of mcmillan's unfolding algorithm, in: Tools and Algorithms for Construction and Analysis of Systems (TACAS), Vol. 1055 of Lecture Notes in Computer Science, Springer, 1996, pp. 87–106.

[41] V. Khomenko, M. Koutny, W. Vogler, Canonical prefixes of Petri net unfoldings, Acta Informatica 40 (2) (2003) 95–118.

[42] F. Elliger, A. Polyvyanyy, M. Weske, On separation of concurrency and conflicts in acyclic process models, in: EMISA, Vol. 172 of LNI, GI, 2010, pp. 25–36.

[43] G. Decker, J. Mendling, Process instantiation, Data & Knowledge Engineering (DKE) 68 (9) (2009) 777–792.

[44] D. Fahland, From scenarios to components, Ph.D. thesis, Humboldt-Universität zu Berlin and Technische Universiteit Eindhoven (2010).

[45] K. L. McMillan, Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in: Computer Aided Verification (CAV), Vol. 663 of Lecture Notes in Computer Science, Springer, 1992, pp. 164–177.

[46] S. Melzer, S. Römer, Deadlock checking using net unfoldings, in: Computer Aided Verification (CAV), Vol. 1254 of Lecture Notes in Computer Science, Springer, 1997, pp. 352–363.

[47] G. Keller, T. Teufel, SAP R/3 Process Oriented Implementation: Iterative Process Prototyping, Addison-Wesley, 1998.

[48] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow patterns, Distributed Parallel Databases 14 (1) (2003) 5–51.

[49] M. Dumas, L. García-Bañuelos, A. Polyvyanyy, Y. Yang, L. Zhang, Aggregate quality of service computation for composite services, in: International Conference on Service Oriented Computing (ICSOC), Vol. 6470 of Lecture Notes in Computer Science, 2010, pp. 213–227.