Maximal Structuring of Acyclic Process Models

Artem Polyvyanyy[1], Luciano García-Bañuelos[2], Dirk Fahland[3] and Mathias Weske[1]

[1]*Hasso Plattner Institute, University of Potsdam, Germany*
[2]*Institute of Computer Science, University of Tartu, Estonia*
[3]*Eindhoven University of Technology, The Netherlands*
Email: Artem.Polyvyanyy@hpi.uni-potsdam.de

This article addresses the transformation of a process model with an arbitrary topology into an equivalent structured process model. In particular, this article studies the subclass of process models that have no equivalent well-structured representation but which, nevertheless, can be partially structured into their maximally-structured representation. The transformations are performed under a behavioral equivalence notion which preserves the observed concurrency of tasks in equivalent process models. The article gives a full characterization of the subclass of acyclic process models that have no equivalent well-structured representation but do have an equivalent maximally-structured one, as well as proposes a complete structuring method. Together with our previous results, this article completes the solution of the process model structuring problem for the class of acyclic process models.

Keywords: Process modeling; structured process model; maximal structuring; model transformation; fully concurrent bisimulation

1. INTRODUCTION

Process models have become the key design artifacts in Business Process Management (BPM) [1, 2]. They are used to design new processes, document existing processes, discuss process improvement, and last but not least, drive process executions in Process-Aware Information Systems (PAIS) [3]. Their dual role as executable code *and* as means of communication among process stake holders has raised debates and concerns about how a particular process should be represented in a process model. In summary, execution and analysis favor, or even demand, specific structural properties which are hard to adhere to when creating process models. This renders the transformation of arbitrary unstructured models into equivalent well-structured models a highly relevant problem [4, 5, 6]. This article extends the technique for structuring acyclic process models [6] with an approach for maximal structuring of inherently unstructured process models, i.e., a process model that has no equivalent well-structured representation gets partially structured to the point where it cannot be structured further.

A process model is usually represented as a graph, where nodes stand for tasks or decisions, and edges encode causal dependencies between adjacent nodes. Common process modeling notations, such as the Business Process Model and Notation (BPMN) [7] or Event-driven Process Chains (EPC) [8], allow process models to have almost any topology. Structural freedom allows for a large degree of creativity when modeling. Nevertheless, it is often preferable that models follow certain structural patterns. The arguably most-agreed on structural property of process models is structuredness [4].

*A process model is **(well-)structured**, if 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 fragment of the model between the split and the join forms a single-entry-single-exit (SESE) process component; otherwise the model is **unstructured**.*

Figure 1(a) shows a process model. In the figure, every dotted box defines a process *component* composed from the arcs that are inside or intersect the box. Split u has a corresponding join z; together they define SESE component $R1$. Yet, splits v and w have no

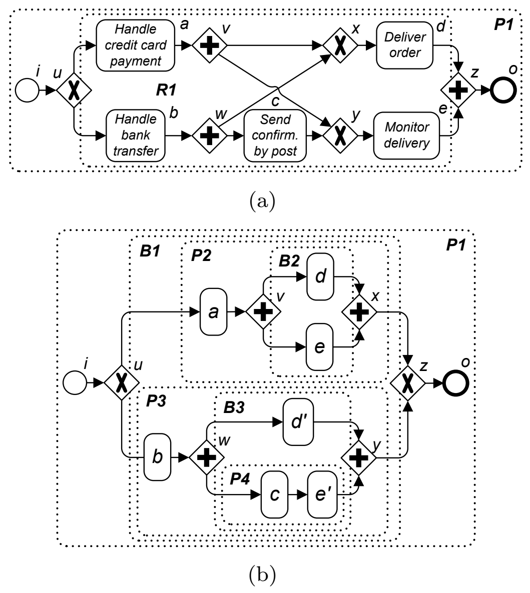


FIGURE 1. (a) Unstructured model, and (b) its equivalent well-structured version

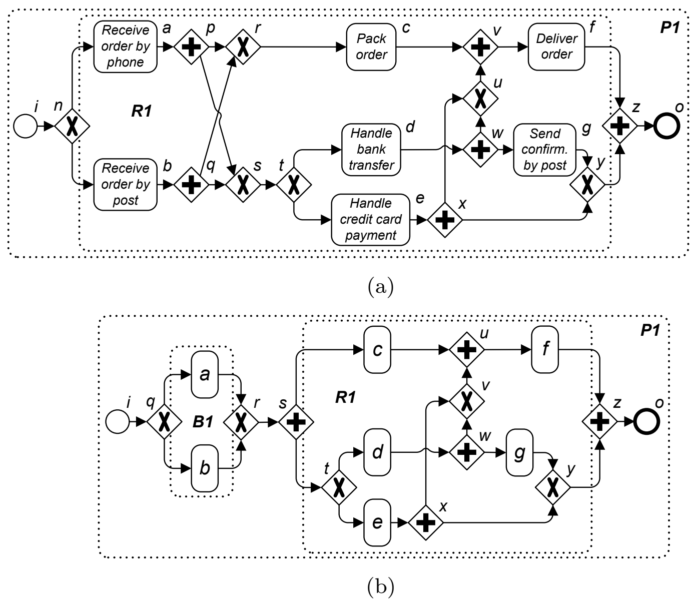


FIGURE 2. (a) Unstructured model, and (b) its equivalent maximally-structured version

corresponding joins and, thus, the model in Figure 1(a) is unstructured. Figure 1(b) shows a well-structured process model, which is equivalent to the model in Figure 1(a). Every split of the model has a corresponding join; split u has corresponding join z, v has x, and w has y; they define three SESE components $B1$, $B2$, and $B3$, respectively. Note that Figure 1(b) uses short-names for tasks $(a, b, c \ldots)$, which appear next to each task in Figure 1(a).

The motivations for well-structured process modeling are manifold: Structured models are easier to layout [9, 10]. It has been empirically shown that structured process models are easier to comprehend and tend to have fewer errors than unstructured ones [11]. By transforming unstructured process models to structured ones, one extends the applicability of process analysis techniques which are only applicable for structured models [12, 13] to a larger class of models and improves translations between models captured in different languages [14, 15, 16]. Structured process models are preferred in the context of refactoring large process model repositories [17, 18, 19]. Structured process models are better suited for optimization [20]. Consequently, some process modeling languages urge for structured modeling, e.g., Business Process Execution Language (BPEL) [21] and ADEPT [22, 23]. However, a modeling methodology that confines itself to "structured" languages faces certain limitations: (i) There exist process models that have no equivalent well-structured version [4]; this simply means that certain processes cannot be modeled. (ii) Structured modeling implies design time constraints and, thus, limits creativity of process model developers. Process modeling is carried out by humans who undergo creative practices. Different people can come up with different solutions to capture the very same process, for example the two process models in Figure 1. Enforcement of structured modeling can lower productivity of a developer. We advocate for modeling without limitations: The modeling language should provide process model developers with a maximal degree of structural freedom to describe processes. Afterwards, scientific methods can propose (upon request and whenever possible) alternative formalizations that are "better" structured, preferably well-structured. Therefore, one should be allowed to specify the model as in Figure 1(a) and, if requested, the equivalent structured model, as in Figure 1(b), should be constructed automatically. Alternatively, unstructured models can result from model synthesis techniques such as process mining [24]; structuring can then be applied to improve the readability of the mined models.

In a previous work [6], we proposed a technique to automatically transform acyclic process models with arbitrary topologies into equivalent well-structured models. The structuring is accomplished under a strong notion of behavioral equivalence, called *fully concurrent bisimulation* [4, 6]. As an outcome, the resulting well-structured models describe the same share of concurrency as the original unstructured models. It was shown in [4] (by means of a single example) and confirmed in [6] (for the general case of acyclic models) that there exist process models that do not have an equivalent well-structured representation. Figure 2(a) is an example of a model which has no equivalent well-structured version. Though not completely structurable, the model can be partially structured to result in its *maximally-structured* version. Intuitively, maximal structuring should be understood as follows.

*A process model P is **maximally-structured**, if every process model that is equivalent to P has at most the same number of SESE components defined by pairs of split and join nodes as P.*

Here, SESE components composed of a single arc or a sequence of arcs, are ignored during comparison as they, in isolation, do not contribute to structural complexity. Figure 2(b) shows a maximally-structured version of the model in Figure 2(a). Though not well-structured, the model in Figure 2(b) is "better" structured than its equivalent model in Figure 2(a). The maximally-structured model has two SESE components $B1$ and $R1$ defined by the split-join pairs (q, r) and (s, z), respectively, as compared to one component $R1$ defined by the split-join pair (n, z) in Figure 2(a). Moreover, there exists no process model which describes the same process as the models in Figure 2 and shows more structure, that is, has more SESE components.

After the initial investigations in [25], this article gives for the first time a complete solution to the problem of maximal structuring of acyclic process models. We characterize the class of acyclic process models which do not have an equivalent well-structured representation, but which can, nevertheless, be maximally-structured; and we provide a complete structuring method. Together with the results in [6], this article completes the structuring technique for the class of acyclic process models.

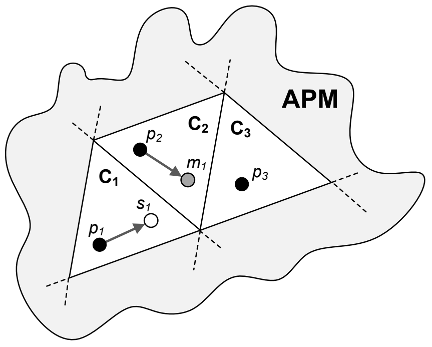


FIGURE 3. Behavioral equivalence relation on the set of all acyclic process models and its three equivalence classes

Figure 3 visualizes the overall setting of the process model structuring problem. In that figure, region APM represents the set of all Acyclic Process Models, where every dot inside the region represents a model: p_1, p_2, and p_3 represent process models which are subjects for structuring. The behavioral equivalence relation is an equivalence relation on APM [26]. C_1, C_2, and C_3 are the equivalence classes of the behavioral equivalence relation that contain models p_1, p_2, and p_3, respectively. Every equivalence class contains the set of all behaviorally equivalent process models. The task of structuring is to answer the question whether an equivalence class contains a model which is "better" structured than a given one from the same class, preferably a well-structured one. In our example, class C_1 contains well-structured model s_1 (e.g., C_1 is the class which contains the models from Figure 1), class C_2 contains no well-structured but a maximally-structured model m_1 (e.g., C_2 is the class with the models from Figure 2), whereas class C_3 contains no models which exhibit more structure than model p_3 and, thus, p_3 is *inherently unstructured*. In this article, we give an answer to the problem of structuring acyclic models.

The remainder of this article proceeds as follows: The next section discusses related work on structuring sequential programs and process models. Afterwards, Section 3 gives preliminary definitions which we shall use later to convey our findings. In Section 4, we formally define the "well-structured" property of process models. Section 5 discusses the structuring technique proposed in [6]. The technique is summarized as a chain of transformations. We define for the first time the notion of the proper complete prefix unfolding which was sketched in [6] and which is essential for obtaining sufficient behavioral information to allow maximal structuring. Section 6 motivates and defines the "maximally-structured" property of process models. Afterwards, Section 7 extends the structuring technique from Section 5 for maximal structuring of process models that do not have an equivalent well-structured representation. Section 8 presents an empirical evaluation of the structuring method on a set of process models taken from industrial practice. Finally, we draw conclusions.

2. RELATED WORK

Our setting is close to the problem of structuring flowcharts and sequential programs, which has been extensively studied for years. In one of his letters, Edsger W. Dijkstra started a discussion with a provocative title "Go To Statement Considered Harmful" [27]. In the absence of Go To statements, programs are composed of structured flow constructs only, situation that corresponds to our notion of well-structured process models without concurrency constructs. The main idea communicated in the letter is that in the context of sequential programs the Go To statement should be abolished from all "higher level" programming languages. Since that time, many replies to the letter supported or rejected the statement of Dijkstra, for instance [28, 29, 30, 31], with no side being able to provide sound arguments to disarm one another. The partial resolution of the conflict became possible due to many works on formal techniques for translating unstructured programs with Go To statements into equivalent structured programs [32, 33, 34, 35]. The main outcome of these endeavors is that any sequential program can be structured with only one control flow pattern, viz. forward jump from a loop, requiring introduction of fresh control variables in structured programs [34]; other flow patterns can be structured by employing code duplications and structured control flow constructs like `if-then-else` or `while-do`.

The results on structuring sequential programs do not hold for process models which comprise concurrency. One of the earliest studies on the problem of structuring process models is that of Kiepuszewski et al. [4]. The authors showed that not all acyclic process models can be structured by putting forward a counter-example, which essentially boils down to the one in Figure 4 (also known as Z-structure, due to the configuration of causal relations between the tasks). The authors showed that there is no well-structured process model that is equivalent to this one under the fully concurrent bisimulation equivalence notion. They do explore some causes of unstructuredness, but neither give a full characterization of the class of models that can be structured, nor do they define any automated transformation.

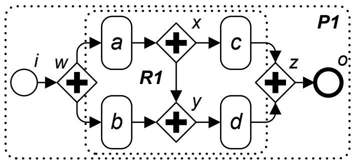


FIGURE 4. Process model (Z-structure)

Some work has been devoted to the characterization of sources of unstructuredness in process models. In [5], the authors present a taxonomy of unstructuredness, that covers acyclic and cyclic models. The taxonomy is based on the notion of improper nesting and mismatched pairs. The taxonomy allows to analyze unstructured process models, determine whether they are well-behaved, and whether they can be transformed into equivalent structured models. However, the taxonomy is incomplete, as it does not cover all possible cases of process models that can be structured. Besides, the authors do not define an automated algorithm for structuring unstructured process models.

Other methods simply reuse techniques for structuring flow charts and sequential programs in order to partly structure process models. [36] proposes a method for structuring sequential parts of a process model based on Go To program transformations, and extends this method to process graphs where concurrent parts are already structured. This method cannot deal with process models which comprise unstructured concurrent threads of control. A similar remark applies to [37], where authors concentrate on structuring of unstructured cyclic flows. In [38], a translation from (unstructured) Petri nets to (structured) BPEL processes is proposed. While the proposed method can handle unstructured concurrent threads of control, it does so by directly expressing them in terms of BPEL's `flow` activity and `links`. Put differently, the method identifies the already structured parts of the process model, but provides no means for structuring the unstructured parts.

TABLE 1. Contribution of this article ($\star$) in the light of our previous work and related work on fundamental structuring techniques; – (no structuring), $\circ$ (partial structuring), + (complete structuring)

	[34]	[4]	[5]	[6]	$\star$
xor (flow charts/programs)	+	-	-	-	-
and (well-struct.)	-	$\circ$	$\circ$	+	+
and (inh. unstr.)	-	$\circ$	$\circ$	+	+
and (maximal)	-	-	-	-	+
xor/and acyclic (well-struct.)	-	$\circ$	$\circ$	+	+
xor/and ac. (inh. unstr.)	-	$\circ$	$\circ$	+	+
xor/and ac. (maximal)	-	-	-	-	+
xor/and cyclic	-	$\circ$	$\circ$	-	-

In [39], the authors outline a classification of process components (parts of process models) using *region trees*. The authors mention that region trees can be employed to transform unstructured process models into well-structured ones, however they do not provide a structuring method, even for acyclic models. In [40], the authors study the influence of "hidden" unstructuredness in process models on their correctness.

Table 1 summarizes related work on structuring techniques [34, 4, 5, 6] and illustrates contributions of this article. Columns of the table correspond to different techniques, whereas rows correspond to different levels of process model expressiveness and its structural properties. The structuring problem is solved for flow charts/sequential programs (*xor* logic, acyclic or cyclic) by compiler techniques, e.g., [34], see the "+" sign in the first row, first column. First *partial* solutions for structuring of process models with concurrency (*and* only and mixed *xor/and* logic, acyclic or cyclic) were proposed in [4, 5]. These two techniques took different approaches to classify sources of unstructuredness, but they do not provide the complete characterization, see the "$\circ$" signs in the table. The technique proposed in [6] completely structures acyclic process models with *xor/and* logic that have a well-structured representation and it is capable of recognizing inherently unstructured process models. All mentioned techniques fail on structuring models with *xor/and* logic that are inherently unstructured but have parts that can be structured, i.e., they fail to achieve maximal structuring. The technique proposed in this article addresses this problem for acyclic models. Note that *xor* only structuring, e.g., [34], is reused by other techniques including the one in this article, whenever components with *xor* only logic occur, see the "-" signs in the first row and the first column of the table.

To sum up, to the best of our knowledge, existing techniques approach structuring of process models with concurrency rather superficially. Existing structuring techniques either drop the requirement of preserving concurrency described in the unstructured process model, or the "problematic" parts of the process model are not structured at all and, hence, unstructuredness remains in the resulting process model. The technique proposed in this article allows to overcome these shortcomings.

3. PRELIMINARIES

In this section, we introduce some formal notions that will be used later to convey findings.

3.1. Process Models and Nets

In the Introduction, process models were presented in a rather informal way. This section introduces all subsequently required formal notions on process models.

Definition 3.1 (Process model). A *process model* $P = (A, G, C, type, \mathcal{A}, \mu)$ has a non-empty set A of *tasks*, a set G of *gateways*, $A \cap G = \varnothing$, and a set $C \subseteq (A \cup G) \times (A \cup G)$ of *control flow* arcs of P; $type : G \to \{xor, and\}$ assigns to each gateway a type; $\mu : A \to \mathcal{A}$ assigns to each task a *name* from $\mathcal{A}$, $\tau \in \mathcal{A}$.

$A \cup G$ are the *nodes* of P; a node $x \in A \cup G$ is a *source* (*sink*), iff $\bullet x = \varnothing$ ($x\bullet = \varnothing$), where $\bullet x$ ($x\bullet$) stands for the set of immediate predecessors (successors) of x. We assume P to have a single source and a single sink; both are tasks. For a task $a \in A$, if $\mu(a) \neq \tau$, then a is *observable*; otherwise, a is *silent*. Every node of P is on a path from the source to the sink. Each task $a \in A$ has at most one incoming and at most one outgoing arc, i.e., $|\bullet a| \leq 1 \wedge |a\bullet| \leq 1$. Each gateway $g \in G$ is either a *split* ($|\bullet g| = 1 \wedge |g\bullet| > 1$) or a *join* ($|\bullet g| > 1 \wedge |g\bullet| = 1$).

We adopt a notation similar to BPMN for visualizing process models as shown in Fig. 1 and 2: a round rectangle represents a task, the name is inscribed; a 45°-rotated square represents a gateway, an inscribed × (+) denotes an *xor* (*and*) gateway. The round nodes i and o in Figure 1(a) are silent tasks. Their role is rather technical: to ensure the existence of the source and sink tasks. The semantics of process models is defined by a mapping to Petri nets.

Definition 3.2 (Petri net). A *Petri net*, or a *net*, $N = (P, T, F)$ has finite disjoint sets P of *places* and T of *transitions*, and the *flow* relation $F \subseteq (P \times T) \cup (T \times P)$.

For a node $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ is the *preset*, whereas $x\bullet = \{y \mid (x, y) \in F\}$ is the *postset* of x; $Min(N)$ denotes the set of nodes of N with an empty preset. A node $x \in P \cup T$ is an *input* (*output*) node of a node $y \in P \cup T$, iff $x \in \bullet y$ ($x \in y\bullet$). For $X \subseteq P \cup T$, let $\bullet X = \bigcup_{x \in X} \bullet x$ and $X\bullet = \bigcup_{x \in X} x\bullet$. For a binary relation R (e.g., F or C), we denote by R^+ the transitive closure, and by R^* the reflexive and transitive closure of R.

In the following, we assume that all nets are *T-restricted*, that is, every transition of a net has at least one input and at least one output place. If this is not the case, we assume the natural completion of the net: the net gets modified so that a transition without an input (output) place gets a single input (output) place.

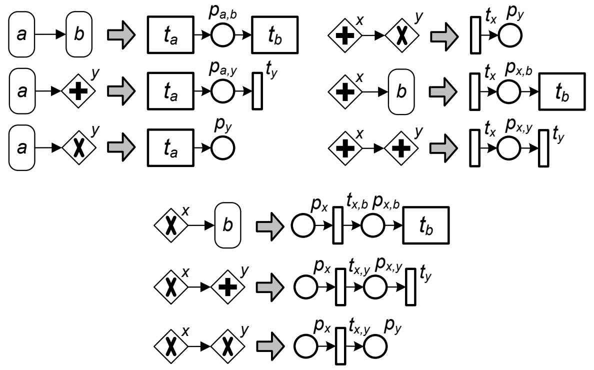


FIGURE 5. Mapping process models to nets

Often, it is useful to distinguish between observable and silent transitions of a net; this distinction can be made formal by *labeling* transitions. A *labeled* net $N = (P, T, F, \mathcal{T}, \lambda)$ consists of a net (P, T, F) and a function $\lambda : P \cup T \to \mathcal{T}$ that assigns each node a *label* from $\mathcal{T}$, $\tau \in \mathcal{T}$. If $\lambda(t) \neq \tau$, then t is *observable*; otherwise, t is *silent*. We require all places of a labeled net to be silent, i.e., $\forall p \in P : \lambda(p) = \tau$.

Petri nets have precise execution semantics grounded on formal notions of states and state transitions, which are defined in terms of a "token game." A state of a net is represented by a *marking* which describes a distribution of *tokens* on the net's places. Whether a transition is *enabled* at a marking depends on the tokens in its input places. An enabled transition can *occur*, which leads to a new marking of the net.

Technically, we identify the flow relation F with its characteristic function on the set $(P \times T) \cup (T \times P)$. Then, the semantics of nets can be formalized as follows.

Definition 3.3 (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 natural number $M(p)$ of *tokens* in p; $\mathbb{N}_0$ is the set of natural numbers including 0. $[p]$ 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 a transition $t \in T$ and a marking M of N, t is *enabled* at M, written $M[t\rangle$, iff $\forall p \in \bullet t : M(p) \geq 1$.
- If $t \in T$ is enabled at M, then t can *occur*, which leads to a new marking M' and the *step* $M[t\rangle M'$ of N with $M'(p) = M(p) - F(p, t) + F(t, p)$, $p \in P$.
- 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.
- A marking M_n is *reachable* in (N, M_0), written $M_n \in [N, M_0\rangle$, iff there exists a *run* of N, i.e., a sequence $M_0[t_0\rangle M_1[t_1\rangle \ldots M_n$ of steps of N that reaches M_n.

In this article, the structural subclass of *free-choice* nets is particularly relevant to us. The free-choice property guarantees that if two transitions share an input place, then every reachable marking of the system enables either both of these transitions or none of them [41, 42].

Definition 3.4 (Free-choice net). A net $N = (P, T, F)$ is *free-choice*, iff $\forall\ p \in P, |p\bullet| > 1 : \bullet(p\bullet) = \{p\}$.

Every process model (Definition 3.1) can be mapped to a *corresponding* labeled free-choice net with a special structure, called WF-net [43, 6].

Definition 3.5 (WF-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 $i \in P$, with $\bullet i = \varnothing$, N has a dedicated *sink* place $o \in P$, with $o\bullet = \varnothing$, 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 net system (N, M_0), where $M_0 = [i]$.

The results of this article are independent of the specific mapping between process models and nets. Subsequently, we shall employ the mapping from [6]; a process model is translated into a net by transforming each of its control flow arcs to the corresponding net fragment as shown in Figure 5.

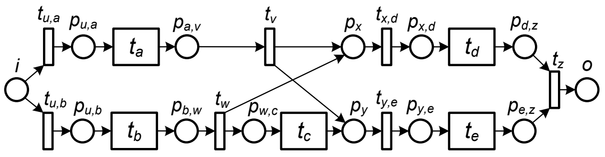


FIGURE 6. WF-net that corresponds to component $R1$ of the process model in Figure 1(a)

The WF-net in Figure 6 corresponds to component $R1$ of the process model in Figure 1(a). Each task of the model is mapped to a transition with its name as a label, for instance $\lambda(t_a)$ = "*Handle credit card payment*". The empty boxes represent silent transitions labeled τ. Note that silent tasks must be mapped to silent transitions (not shown in Figure 5). The execution semantics of the resulting net system[4] defines the semantics of the process model.

Soundness and safeness are basic properties of WF-systems [44]. 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 the same place.

Definition 3.6 (Liveness, Safeness, Soundness).

- A system (N, M_0) is *live*, iff for every reachable marking $M \in [N, M_0\rangle$ and $t \in T$, 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) is *safe*, iff $\forall\ M \in [N, M_0\rangle\ \forall\ p \in P : M(p) \leq 1$.
- A WF-system (N, M_i) with $N = (P, T, F)$ is *sound*, iff the short-circuit system $(N^\star, M_i)$ is live and bounded.

In our work, we require process models to be sound, with the intuition that a process model is sound, iff its corresponding WF-system is sound. As a sound free-choice WF-system is guaranteed to be safe [45], the rest of the article deals with sound and safe process models.

3.2. Unfoldings

An unfolding of a net system is another net that explicitly represents all runs of the system in a possibly infinite, tree-like structure [46, 47]. In [48], McMillan proposed an algorithm to construct a *finite* initial part of the unfolding, which contains full information about the reachable markings of the system, viz. a *complete prefix unfolding*. Next, we present main notions of the theory of unfoldings, starting with ordering relations between pairs of nodes in a net.

Definition 3.7 (Ordering relations).
Let $N = (P, T, F)$ be a net and let $x, y \in P \cup T$ be its nodes.

- x and y are in *causal* relation, written $x \rightsquigarrow_N y$, iff $(x, y) \in F^+$. y and x are in *inverse causal* relation, written $y \leftsquigarrow_N x$, iff $x \rightsquigarrow_N y$.
- x and y are in *conflict*, $x \#_N y$, iff there exist distinct transitions $t_1, t_2 \in T$, s.t. $\bullet t_1 \cap \bullet t_2 \neq \varnothing$, and $(t_1, x), (t_2, y) \in F^*$. If $x \#_N x$, then x is in *self-conflict*.
- x and y are *concurrent*, $x \parallel_N y$, iff neither $x \rightsquigarrow_N y$, nor $y \rightsquigarrow_N x$, nor $x \#_N y$.

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

In the following we omit subscripts of ordering relations where the context is clear. A structure of an unfolding is given by an *occurrence* net.

Definition 3.8 (Occurrence net).
A net $N = (B, E, G)$ is an *occurrence* net, iff : for all $b \in B$ holds $|\bullet b| \leq 1$, N is acyclic, for each $x \in B \cup E$ the set $\{y \in B \cup E \mid (y, x) \in G^+\}$ is finite, and no $e \in E$ is in self-conflict.

The elements of B and E are called *conditions* and *events*, respectively. Any two nodes of an occurrence net are either in causal, inverse causal, conflict, or concurrency relation [49]. A *branching process* of a system S is an occurrence net where each event represents the occurrence of a transition of S. The *unfolding* of S is simply its largest branching process.

The relation between S and its branching processes 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 nets. A

[4]Note that when we refer to a net without a marking as to a net system, we assume its canonical initial marking that puts one token at every place without incoming arcs and no tokens elsewhere.

homomorphism from N_1 to N_2 is a mapping $h : P_1 \cup T_1 \rightarrow P_2 \cup T_2$, 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 3.9 (Branching process).
A *branching process* of a system $S = (N, M_0)$ is a pair $\beta = (N', \nu)$, where $N' = (B, E, G)$ 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 $e_1, e_2 \in E$ holds if $\bullet e_1 = \bullet e_2$ and $\nu(e_1) = \nu(e_2)$, then $e_1 = e_2$.

The system S is referred to as the *originative* system of a branching process. The branching processes of S are ordered by their *prefix* relation.

Definition 3.10 (Prefix relation).
Let $\beta_1 = (N_1, \nu_1)$ and $\beta_2 = (N_2, \nu_2)$ be two branching processes of a system $S = (N, M_0)$. β_1 *is a prefix of* β_2 if N_1 is a subnet of N_2, such that: if a condition belongs to N_1, then its input event in N_2 also belongs to N_1, if an event belongs to N_1, then its input and output conditions in N_2 also belong to N_1, and ν_1 is the restriction of ν_2 to nodes of N_1.

A maximal branching process of S with respect to the prefix relation is called *unfolding* of the system. In [46], it is shown that every system has an (up to isomorphism) unique unfolding. Finally, a *complete prefix unfolding* of S is a special branching process of S: it is obtained by truncating the unfolding at events where the information about reachable markings starts to be redundant.

Definition 3.11 (Complete prefix unfolding).
Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a branching process of a system $S = (N, M_0)$.

- A *configuration* C of β is a set of events, $C \subseteq E$, such that: (i) C is causally closed, i.e., $e \in C$ implies that for all $e' \in E$, $e' \rightsquigarrow e$ implies $e' \in C$, and (ii) C is conflict-free, i.e., for all $e_1, e_2 \in C$ holds $\neg(e_1 \# e_2)$.
- A *local* configuration of an event $e \in E$ is the set of events that precede e, written $\lceil e \rceil = \{e' \in E \mid e' \rightsquigarrow e\}$.
- A set of conditions of an occurrence net is a *co-set* if its elements are pairwise concurrent. A maximal co-set with respect to inclusion is a *cut*.
- Each finite configuration C of β induces the cut $Cut(C) = (Min(N') \cup C\bullet) \setminus \bullet C$ that represents the reachable marking $Mark(C) = \nu(Cut(C))$ of S.
- β is *complete* if for each reachable marking M of S there exists a configuration C in 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 $C \cup \{e\}$ in β, such that $e \notin C$ and $\nu(e) = t$.
- A partial order $\lhd$ on the configurations of β is an *adequate order* iff (i) $\lhd$ is well-founded, (ii) $C_1 \subset C_2$ implies $C_1 \lhd C_2$, and (iii) If $C_1 \lhd C_2$ and $Mark(C_1) = Mark(C_2)$, then $\lhd$ is preserved for all finite extensions of C_1, i.e., the "future" after $Cut(C_1)$ and $Cut(C_2)$ is isomorphic, see [50] for details.
- An event $e \in E$ is a *cutoff* event induced by $\lhd$, iff there exists a *corresponding* event $corr(e) := e' \in E$ such that $Mark(\lceil e \rceil) = Mark(\lceil e' \rceil)$ and $\lceil e' \rceil \lhd \lceil e \rceil$.
- β is the *complete prefix unfolding* induced by $\lhd$, iff β is the greatest prefix of the unfolding of S that contains no event after a cutoff event.

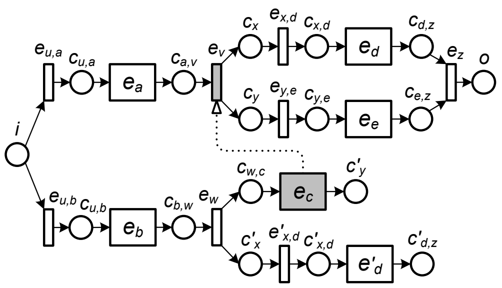


FIGURE 7. Complete prefix unfolding of the system in Figure 6

Figure 7 shows a complete prefix unfolding of the system in Figure 6. We write $c_x, c'_x, c''_x, \ldots$ for conditions that are the occurrences of place p_x; correspondingly for events. In the prefix, event e_c is a cutoff event, whereas event e_v is its corresponding event; this relation is visualized by a dotted arrow. Both events reach the marking $\{x, y\}$ represented by the cuts $\{c_x, c_y\}$ and $\{c'_x, c'_y\}$, respectively. The size of the prefix depends on the "quality" of the adequate order used to perform the truncation. It has been shown that the adequate order proposed in [50] results in more compact prefixes compared to the one suggested in [48].

4. WELL-STRUCTURED MODELS

In the Introduction, we proposed an intuitive definition of the well-structured property of process models. In this section, we give a formal form to that intuition. We propose a structural classification for process models based on the properties of its parse tree. The Refined Process Structure Tree (RPST) is a technique to parse process models into a collection of its fragments, each with a single entry and single exit. The notion of the RPST fragment coincides with the notion of the SESE component used in the intuitive definition of a well-structured process model.

Let $P = (A, G, C, type, \mathcal{A}, \mu)$ be a process model. A *fragment* of P is defined by a pair (V, E) of nodes $V \subseteq A \cup G$ and arcs $E \subseteq C$, such that each arc in E connects only nodes in V. A set $F \subseteq C$ of arcs induces the fragment $P_F = (V_F, F)$ *formed by* F where V_F is the smallest set of nodes such that (V_F, F) is a fragment.

Definition 4.1 (Interior, Boundary, Entry, Exit).
Let $P = (A, G, C, type, \mathcal{A}, \mu)$ be a process model and let $P_F = (V_F, F)$ be a connected fragment of P that is

formed by a set $F \subseteq C$ of control flow arcs.

- A node $x \in V_F$ is *interior* with respect to P_F, iff it is connected only to nodes in V_F; otherwise v is a *boundary* node of P_F.
- A boundary node u of P_F is an *entry* of P_F, iff no incoming edge of u belongs to F or all outgoing edges of u belong to F.
- A boundary node v of P_F is an *exit* of P_F, iff no outgoing edge of v belongs to F or all incoming edges of v belong to F.

A process component is a fragment of a process model with specific boundary nodes.

DEFINITION 4.2 (Process component). Let $P = (A, G, C, type, \mathcal{A}, \mu)$ be a process model and let $P_F = (V_F, F)$ be a connected fragment of P that is formed by a set $F \subseteq C$ of control flow arcs. F is a *process component*, or a *component*, of P, if P_F has exactly two boundary nodes: one is an entry and the other is an exit of P_F.

By definition, the source of a process model is an entry to every component it belongs to, whereas the sink of a process model is an exit from every component it belongs to. In contrast to a fragment, we identify a process component only by its set of arcs. We say that two components F, F' are *nested* if $F \subseteq F'$ or $F' \subseteq F$. They are *disjoint* if $F \cap F' = \varnothing$. If they are neither nested nor disjoint, we say that they *overlap*. The RPST of a process model is a collection of its special components, viz. *canonical* process components.

DEFINITION 4.3 (Canonical process component). Let P be a process model. A process component of P is *canonical*, iff it does not overlap with any other process component of P.

Finally, the RPST of a process model is composed of its canonical process components.

DEFINITION 4.4 (The refined process structure tree). The *refined process structure tree* (*RPST*) of a process model is the set of all its canonical process components.

It follows that any two canonical process components are either nested or disjoint and, thus, they form a compositional containment hierarchy. This hierarchy can be shown as a tree, where the parent of a canonical component F is the smallest canonical component that contains F. Hence, the root of the tree is the entire model, whereas the leaves are the control flow arcs.

Every box with a dotted border in Figure 1 and Figure 2 defines a canonical process component which is composed of the arcs that are inside or intersect the box. Figure 8 shows an alternative visualization of the hierarchies of canonical process components, where every node represents a component and edges hint at the containment relation of components. Figure 8(a) and Figure 8(b) show (somewhat simplified) RPSTs

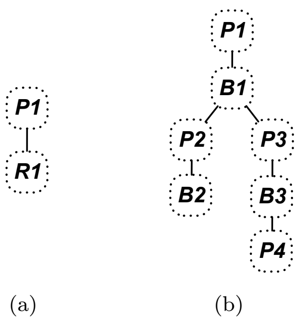


FIGURE 8. The (simplified) RPSTs of the process models of Figure 1

of the process models in Figure 1(a) and Figure 1(b), respectively.

According to [51, Section 3.1], every canonical component of a process model belongs to one out of four structural classes. The names of the structural classes are borrowed from the types of the triconnected components of graphs, which are employed to construct the RPST, see [51] for details.

DEFINITION 4.5 (Component classes). Let F be a component of a process model P.

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

The names of components in Figures 1, 2, and 8 hint at their structural class, e.g., $P1$ is a polygon, $B1$ is a bond, and $R1$ is a rigid component. In these figures, we do not explicitly show *simple* components, i.e., trivial components or polygons composed of two trivial components. Hence, we refer to the respective RPSTs as simplified ones. To exemplify the concepts, we refer to the models in Figure 1. For instance, polygon $P1$ in Figure 1(a) is composed of trivials $\{(i,u)\}$, $\{(z,o)\}$ and rigid $R1$, while bond $B1$ in Figure 1(b) is composed of polygons $P2$ and $P3$. Polygons $\{(u,a),(a,v)\}$, $\{(u,b),(b,w)\}$, $\{(w,c),(c,y)\}$, $\{(x,d),(d,z)\}$, $\{(y,e),(e,z)\}$ are the simple components inside rigid $R1$ in Figure 1(a). For a complete visualization of the RPST, the trees in Figure 8 have to be refined with these simple components.

The RPST of a process model is unique [52]. Moreover, the canonical components of a process model define all its SESE components, where non-canonical SESE components can be derived as subsets of polygons and bonds.

With all of the above, the well-structuredness property of a process model can be defined in terms of its RPST.

Definition 4.6 (Well-structured model). A process model is *(well-)structured*, iff its RPST contains no rigid process component; otherwise the model is *unstructured*.

The process model in Figure 1(b) contains no rigids and is structured, while the process model in Figure 1(a) contains rigid $R1$ and is, therefore, unstructured.

In order to introduce the formal definition of the maximally-structured property of a process model, one needs first to understand the structuring method proposed in Section 5 and its limitations. We return to maximal structuring in Section 6, where we take a closer look at the structuring scenario depicted in Figure 2.

5. STRUCTURING

This section discusses the technique for structuring acyclic process models from [6]. We elaborate further on the technique by proposing the notion of the proper complete prefix unfolding for the first time; we shall see that this type of prefix is essential to achieve *maximal* structuring.

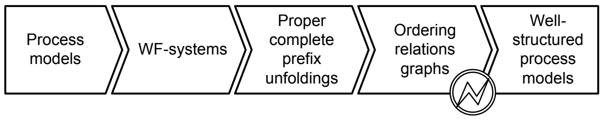

Process models
WF-systems
Proper complete prefix unfoldings
Ordering relations graphs
Well-structured process models

FIGURE 9. Structuring chain, see [6]

Figure 9 shows the chain of phases that collectively compose the structuring technique. A process model first is decomposed into the hierarchy of its process *components*. Each component is a process model by itself and either well-structured or unstructured. An unstructured process component can in some cases be transformed into a well-structured one. For this purpose, the component is translated into a WF-system for which the ordering relations of its tasks are derived from its proper complete prefix unfolding. If the ordering relations have certain properties, the unstructured component can be replaced by a hierarchy of smaller well-structured components that define the same ordering relations. The equivalence of the resulting process model with the original unstructured one is guaranteed by the results presented in [53].

In the following, we present each phase of the structuring in detail. We employ the process model in Figure 1(a) to exemplify all structuring phases. Later, in Section 7, we extend the technique to allow maximal structuring.

5.1. From process models to unfoldings

A process model is well-structured, if and only if its RPST contains no rigid component (Definition 4.6). Therefore, an unstructured process model can be structured by traversing its RPST bottom-up and replacing each rigid component by its equivalent well-structured component. The difficult step is to find this equivalent well-structured component.

The key idea of structuring is to *refine* a rigid component R, i.e., a *node* of the RPST, by a subtree of well-structured RPST nodes which define the same behavioral (ordering) relations between the child RPST nodes of R. The first step for refining R is to compute the ordering relations of R's child nodes. We obtain these by constructing a complete prefix unfolding of R's corresponding WF-system. The complete prefix unfolding captures information about all reachable markings of the originative system, but has a simpler structure as it is an occurrence net (Definition 3.8). To capture all well-structuredness contained in R, the complete prefix unfolding must have a specific shape, called *proper*.

Definition 5.1 (Proper complete prefix unfolding). Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be a branching process of an acyclic system $S = (N, M_0)$.

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

A proper prefix contains all information about well-structuredness, i.e., all paired gateways of splits and joins, in a rigid in the following way: A proper prefix β represents each *xor* split as a condition with multiple post-events; each *xor* join is identified by the post-conditions of a cutoff event e and its corresponding event. The notion of a cutoff event guarantees that β contains every *xor* split and join. An important observation here is that corresponding pairs of *xor* splits and joins are always contained in the same branch of β. An *and* split manifests as an event with multiple post-conditions in β, whereas an *and* join is an event with multiple pre-conditions. The healthiness requirement on cutoff events ensures that concurrency after an *and* split is kept encapsulated: when several concurrent branches are introduced in the unfolding they are not truncated until the point of their synchronization, i.e., the *and* join. Such an intuition supports our goal to derive a well-structured process model, because bonds of a process model that define concurrency must be synchronized in the same branch of the model where they forked.

Figure 10 shows a proper prefix of the system in Figure 6, which corresponds to the rigid component $R1$ in Figure 1(a). Note that in our example, the proper prefix is different from the (non-proper) complete prefix unfolding of the system shown in Figure 7, and happens to be the unfolding of the system. This is not the general case; Figure 11 shows a complete prefix unfolding of the running example in [6] which is proper and smaller than

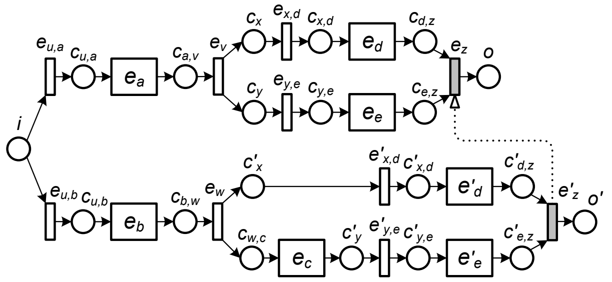


FIGURE 10. Proper complete prefix unfolding of the system in Figure 6

the unfolding. Cutoff event e_c in Figure 7 is not healthy as $Cut(\lceil e_c \rceil) \setminus e_c\bullet = \{c'_x\}$, while $Cut(\lceil e_v \rceil) \setminus e_v\bullet = \emptyset$, where e_v is the corresponding event. The only cutoff event e'_z in Figure 10 is healthy, this results in the proper prefix where all the concurrency introduced by event e_w is synchronized in the same branch of the prefix by event e'_z.

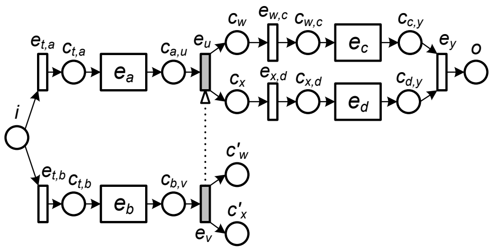


FIGURE 11. Proper complete prefix unfolding

A proper complete prefix unfolding of an acyclic system is clearly finite. For structuring purposes, when computing proper prefixes, we use an adequate order for safe systems proposed in [50]. This adequate order is a total order and results in minimal complete prefix unfoldings, if one only considers information about reachable markings induced by local configurations, which is the case for healthy cutoff events. Thus, the adequate order from [50] always yields minimal proper prefixes of safe acyclic systems, which applies to our case; recall that sound free-choice WF-nets are safe.

5.2. From unfoldings to graphs

The proper complete prefix unfolding of a process component R contains all ordering relations of all children of R. For structuring, R is to be refined into a subtree along these ordering relations. The refinement requires this information to be preserved in a hierarchically decomposable form: an ordering relations *graph*.

DEFINITION 5.2 (Ordering relations graph). Let $\beta = (N', \nu)$, $N' = (B, E, G)$, be the proper prefix of a sound acyclic free-choice WF-system $S = (N, M_i)$, $N = (P, T, F, \mathcal{T}, \lambda)$. Let $x, y \in B \cup E$ be nodes of N'.

- x and y are in *proper causal* relation, written $x \rightarrowtail_{N'} y$, iff $(x,y) \in G^+$ or there exists a sequence $(e_1, \ldots, e_n)$ of healthy cutoff events of β, $e_i \in E$, $1 \leq i \leq n$, $n \in \mathbb{N}$, such that $(x, e_1) \in G^*$, $(corr(e_i), e_{i+1}) \in G^*$ for $1 \leq i < n$, and $(corr(e_n), y) \in G^+$. y and x are in *inverse proper causal* relation, written $y \leftarrowtail_{N'} x$, iff $x \rightarrowtail_{N'} y$.
- Let $\mathcal{R}_{N'} = \{\rightsquigarrow_{N'}, \leftsquigarrow_{N'}, \#_{N'}, \|_{N'}\}$ be the ordering relations of N'. The *proper conflict* relation of N' is $\boxplus_{N'} = \#_{N'} \setminus (\rightarrowtail_{N'} \cup \leftarrowtail_{N'})$. The set $\mathcal{R}'_{N'} = \{\rightarrowtail_{N'}, \leftarrowtail_{N'}, \boxplus_{N'}, \|_{N'}\}$ forms the *proper ordering relations* of N'.
- We refer to (proper) orderign relations $\mathcal{R}$ as *observable*, iff the relations in $\mathcal{R}$ only contain pairs of events that correspond to observable transitions.
- Let $\mathcal{R}'_{N'} = \{\rightarrowtail_{N'}, \leftarrowtail_{N'}, \boxplus_{N'}, \|_{N'}\}$ be the observable proper ordering relations of N'. An *ordering relations graph* $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ of N' has vertices $V \subseteq E$ defined by events of β that correspond to observable transitions of N, arcs $A = \rightarrowtail_{N'} \cup \boxplus_{N'}$, and a labeling function $\sigma : V \to \mathcal{B}$, $\mathcal{B} = \mathcal{T} \setminus \{\tau\}$ with $\sigma(v) = \lambda(\nu(v))$, $v \in V$.

Again, in the following we omit subscripts of ordering relations where the context is clear. An ordering relations graph of a proper complete prefix unfolding captures minimal and complete information about the ordering relations of events that correspond to observable transitions of the originative system. Figure 12(a) visualizes the ordering relations graph of the proper complete prefix unfolding in Figure 10.

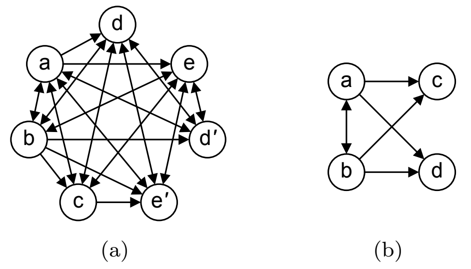


FIGURE 12. Ordering relations graphs: (a) of the proper prefix in Figure 10, (b) of the proper prefix in Figure 11

In the figure, vertices represent events of the proper prefix, e.g., vertex a represents event e_a. Due to a design decision, arcs of an ordering relations graph encode proper causal and proper conflict relations. The graph denotes that a and d are in proper causal relation (single-headed arrow from a to d), b and d are in proper conflict (two-headed arrow between a and b), whereas d and e are concurrent (no arrow). The proper causal relation updates the causal relation of the proper prefix to overcome the effect of unfolding truncation. As an outcome, $e_b \rightarrowtail e_c$ and $e_b \rightarrowtail e_d$ in the proper prefix in Figure 11, which is also reflected in the ordering relations graph in Figure 12(b).

5.3. From graphs to process models

The ordering relations graph not only encodes the ordering relations, it also inherits *all* information about well-structuredness from the proper prefix as pairing of gateways is preserved. The structuring technique in [6] proceeds by parsing the graph into a hierarchy of subgraphs that encode ordering relations of well-structured components. The thereby discovered hierarchy of subgraphs is then used to refine the originative rigid component into a subtree. As shown in [6], each subgraph corresponds to the notion of a *module* of the modular decomposition of a directed graph [54] – thus discovering well-structuredness in the relations of an unstructured process component.

Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph. A *module* $M \subseteq V$ in $\mathcal{G}$ is a non-empty subset of vertices of $\mathcal{G}$ that are in uniform relation with vertices $V \setminus M$: if $v \in V \setminus M$, then v has directed edges to all members of M or to none of them, and all members of M have directed edges to v or none of them do. However, two vertices $v_1, v_2 \in V \setminus M$, $v_1 \neq v_2$, can have different relations to members of M. Moreover, the members of M and $V \setminus M$ can have arbitrary relations to each other [54]. For example, $\{d, e\}$ is a module in Figure 12(a), as both d and e are in the inverse proper causal relation with a, and in proper conflict with every other vertex of the graph.

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. Module M_1 is *strong*, iff there exists no module M_2 of $\mathcal{G}$, such that M_1 and M_2 overlap.

Definition 5.3 (The modular decomposition tree). The *modular decomposition tree* (MDT) of an ordering relations graph is the set of all its strong modules.

The modular decomposition substitutes each strong module of $\mathcal{G}$ by a new vertex and proceeds recursively. The result is the MDT which is a unique canonical rooted tree. The MDT of a directed graph can be computed in linear time [54].

Now, a rigid process component R of an RPST can be structured by refining R in the RPST to a subtree T_R. The root of T_R is child of R's parent, each child of R is attached to a leaf of T_R, the nodes of T_R are defined by the modules of the MDT of R's ordering relations graph. The class of a node of T_R is determined by the characteristics of its defining MDT module. Every module of the MDT belongs to one out of four structural classes.

Definition 5.4 (Module classes).
Let M be a module of an ordering relations graph $\mathcal{G}$.

- M is a *trivial* module, iff M is singleton.
- 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 edge 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. If the subgraph is complete, then we refer to M as *xor* complete. If the subgraph is edgeless, then we refer to M as *and* complete.
- M is a *primitive* module, iff M is neither trivial, nor complete, nor linear. A primitive module is *concurrent*, iff it contains a pair of vertices not connected by an edge in $\mathcal{G}$.

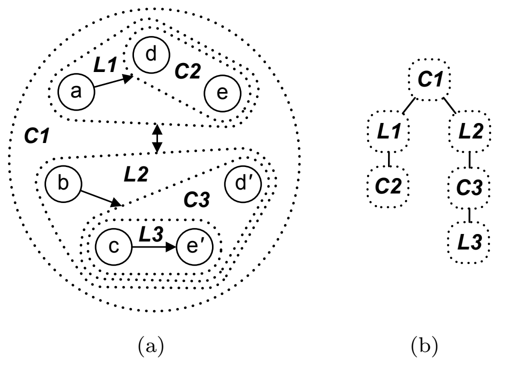


FIGURE 13. Two different visualizations of the MDT of the ordering relations graph in Figure 12(a)

Figure 13(a) shows the MDT of the ordering relations graph in Figure 12(a). Besides the trivial modules, the MDT contains linear modules $L1$, $L2$, and $L3$, *xor* complete module $C1$, and *and* complete modules $C2$ and $C3$. Module $C1$ is the root module, whereas trivial modules are leafs of the MDT. In the figure, each area enclosed by a dotted line defines a module composed of the modules inside the area. Edges between modules represent relations between every pair of vertices composed of one vertex from each of the adjacent modules. Module names hint at their class, e.g., $C1$ is complete and $L1$ is linear. Figure 13(b) shows an alternative visualization of the MDT given as a tree-like structure, where nodes represent non-trivial modules and edges encode containment relation.

An acyclic process model has an equivalent well-structured model, if its ordering relations graph contains no concurrent primitive module. According to [6], the behavior captured by other module classes can be expressed by well-structured process components. A trivial module corresponds to a task. A linear module corresponds to a polygon component. An *and* (*xor*) complete module corresponds to a bond with *and* (*xor*) gateways as entry and exit nodes. Finally, a primitive module without concurrency can be structured using standard compiler techniques [34]. A well-structured process model constructed in such a way is fully concurrent bisimilar with the original unstructured model.

Given all of the above, Algorithm 1 summarizes the structuring technique. Algorithm 1 traverses the MDT of an ordering relations graph of a rigid process component P and constructs for each encountered module a process component from components that correspond to its child

Algorithm 1: Structuring Acyclic Process Model

```
Input: An acyclic process model $P$
Output: An equivalent structured process model
1  Construct WF-net $N$ that corresponds to $P$
2  Construct proper complete prefix unfolding $\beta$ of $N$
3  Construct ordering relations graph $\mathcal{G}$ of $\beta$
4  Compute $\mathcal{M}$ – the MDT of $\mathcal{G}$
   // Construct process model $P'$ by traversing $\mathcal{M}$ in postorder
5  foreach module $m$ of $\mathcal{M}$ do
6      switch class of $m$ do
7          case $m$ is trivial do
8              Construct a task
9          case $m$ is and complete do
10             Construct an and bond
11         case $m$ is xor complete do
12             Construct a xor bond
13         case $m$ is linear do
14             Construct a trivial or polygon
15         case $m$ is non-concurrent primitive do
16             Construct a structured component
               using compiler techniques, e.g., [34]
17         otherwise do
18             FAIL
19 return $P'$
```

modules. The resulting hierarchy of components is the subtree that refines the rigid component P. In our example, the RPST of the process model in Figure 1(a) contains one rigid $R1$, see also Figure 8(a). $R1$ can be refined by a subtree of RPST nodes that correspond to modules of the MDT in Figure 13(b) to result in the RPST shown in Figure 8(b). First, modules $C2$ and $L3$ can be used to construct components $B2$ and $P4$ in Figure 1(b). Next, modules $L1$ and $C3$ must be employed to construct components $P2$ and $B3$. Then, module $L2$ results in component $P3$. Finally, module $C1$ results in component $B1$.

6. MAXIMALLY-STRUCTURED MODELS

In this section, we give a formal definition of the maximally-structured property of process models. Unlike well-structuredness, maximal structuredness is founded on a relation *between the RPST and the MDT of the model.* We gain insights into this relation by applying the structuring technique from Section 5 on the process model of Figure 2(a). To simplify language, in the following, we often skip several phases of the structuring technique when referring to the concepts of interest. For example, the "MDT of a process model" is the MDT obtained by mapping the model to a WF-net, constructing the proper prefix, then its ordering relations graph and, finally, computing the MDT of the

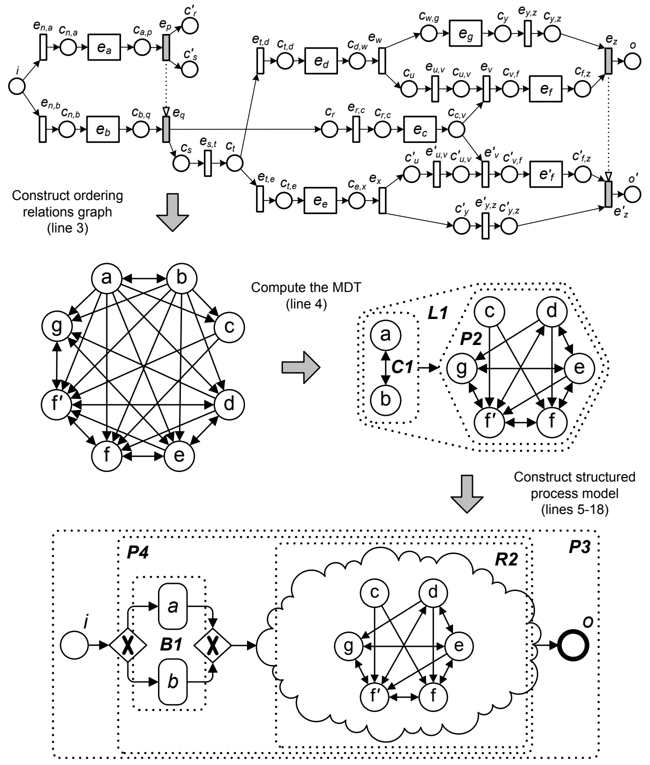

Construct ordering relations graph (line 3)
Compute the MDT (line 4)
Construct structured process model (lines 5-18)
L1
P2
C1
P4
R2
P3
B1
i
o

FIGURE 14. Structuring the process model in Figure 2(a)

graph.

Figure 14 visualizes transformation steps 3–18 of Algorithm 1 applied on the process model of Figure 2(a). The proper prefix of the model (shown at the top of the figure) is the result of line 2 of the algorithm. Next (follow the arrow directions), the algorithm constructs the ordering relations graph and its MDT (lines 3–4). Finally, the algorithm attempts to construct a process model from the MDT (lines 5–18). As the MDT contains primitive module $P2$, the algorithm fails at line 18. However, the MDT exhibits some information on structuredness of the ordering relations: it contains linear module $L1$ and *xor* complete module $C1$. These modules can be used to construct structured components $P4$ and $B1$ shown at the bottom of Figure 14. If one is able to synthesize a process component which demonstrates the ordering relations captured in primitive module $P2$, then one can construct a model which exhibits more structuredness than the original one. Note that in general a primitive module defines a subgraph of the ordering relations graph and, hence, the topology of the corresponding process component in the RPST cannot be deduced from the original model.

In a maximally-structured model every module of its MDT must have a corresponding component in its RPST.

Definition 6.1 (Maximally-structured model). A process model is *maximally-structured*, iff there exists isomorphism between the RPST and the MDT of the process model, where simple components of the RPST and trivial modules of the MDT are ignored, which assigns to every polygon component a linear module, to every bond a complete, to every rigid a primitive, and all primitives in the MDT are concurrent.

Recall that a simple component is either a trivial process component, or a polygon composed of two trivials (Section 4). Due to the properness of the complete prefix unfolding, the ordering relations graph of a given process model and, hence, its MDT are unique. Consequently, each process model has exactly one unique fully concurrent bisimilar maximally-structured version.

Figures 15(a) and 15(c) show the RPST of the model in Figure 2(a) and the subtree of the RPST of the model at the bottom of Figure 14, respectively, without simple components. In Figure 15(b), one can see the MDT from the structuring scenario in Figure 14, without trivial modules. This MDT is isomorphic to the RPST in Figure 15(c): Linear $L1$ can be mapped to polygon $P4$. Complete $C1$ can be mapped to bond $B1$; primitive $P2$ can be mapped to rigid $R2$. In the final RPST of the maximally-structured version of Figure 2(a), polygon $P4$ must be merged with polygon $P3$ (as $P4$ is a subsequence of $P3$ and thus not canonical), see [51] for details. Note that Definition 6.1 is based on the implicit assumption that the isomorphism between both trees canonically extends to an isomorphism that maps process components with tasks onto modules of events with the same multisets of names/labels.

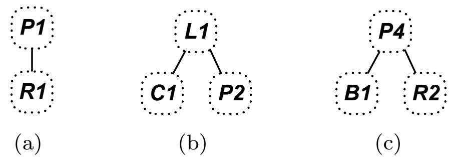


FIGURE 15. (a) The RPST of the model in Figure 2(a), (b) the MDT and (c) the subtree of the RPST from the structuring scenario in Figure 14, all simplified

In the next section, we explain a technique for maximal structuring of process models, which essentially boils down to the technique for synthesis of a process component from a given ordering relations graph.

7. MAXIMAL STRUCTURING

The open problem from Section 5 is to structure a given rigid process component into an equivalent maximally-structured process component R. In the light of Section 6, R has this property, iff (i) all primitive modules in the MDT of R's ordering relations graph are concurrent, and (ii) there exists a bijection between non-singleton modules of the MDT and non-simple components of the RPST which assigns to each primitive module a rigid component, to each complete a bond, and to each linear a polygon. The maximal structuredness of R follows from the maximality of the modular decomposition: the ordering relations graph of R inherits all information about well-structuredness from the proper prefix of R, and the MDT maximizes modules with a well-structured representation because of the decomposition into strong modules. If a concurrent primitive module M has well-structured child modules, then these modules are maximal again within M. Only the relations within M have no structured representation as process model. That is, M is minimized by maximizing structuredness around and inside M. This yields a technique for maximal structuring: one must be able to synthesize a process component that exhibits the ordering relations described in M. Such a technique would allow to define unstructured process model topologies by mapping hierarchies of modules onto hierarchies of process components in Algorithm 1, e.g., the primitive module in Figure 14 onto the rigid component in Figure 2(b). The resulting process model would be maximally-structured.

In this section, we propose a solution to the synthesis problem, i.e., given an ordering relations graph (a module of an MDT) we synthesize a process model (a component of the RPST) that realizes the relations described in the graph. The central idea is to first construct from the ordering relations graph an occurrence net that is interpreted as the unfolding of the process model and that exhibits the same ordering relations. *Refolding* this unfolding then yields the process model we wish to synthesize. The entire procedure requires several phases that employ results of domain- and net theory [49], and on folding prefixes of systems [55]. The procedure extends the structuring approach of Figure 9 and is shown in Figure 16. The reconstruction of the occurrence net is explained in detail in Sections 7.1–7.3, and the refolding to a process model in Sections 7.4–7.6.

7.1. From graphs to partial orders

This section describes a translation from an ordering relations graph to a partial order of information. The partial order is an alternative formalization of the behavior captured in the graph. The elements of the partial order essentially correspond to the configurations of a branching process (Definition 3.11).

The ordering relations graph in Figure 17 is a module from the running example of Section 6. The graph is a primitive module with all types of relations; f and f' represent events with the same label.

First, we give some definitions from the theory of partially ordered sets (posets) [49]. Let $(D, \sqsubseteq)$ be a poset. For a subset X of D, an element $y \in D$ is an upper (lower) bound of X, iff $x \sqsubseteq y$ ($x \sqsupseteq y$), for each element $x \in X$. An element $y \in D$ is a greatest (least) element, iff for each element $x \in D$ holds $x \sqsubseteq y$ ($x \sqsupseteq y$). An element $y \in D$ is a maximal (minimal) element, iff there exist no element $x \in D$, such that $y \sqsubset x$ ($x \sqsubset y$); D_{max} and D_{min} denote the sets of maximal and minimal

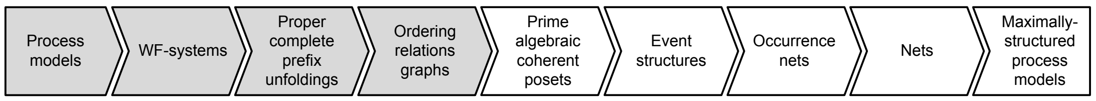

Process models
WF-systems
Proper complete prefix unfoldings
Ordering relations graphs
Prime algebraic coherent posets
Event structures
Occurrence nets
Nets
Maximally-structured process models

FIGURE 16. An extension of the structuring chain of Figure 9

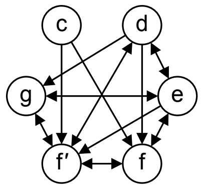


FIGURE 17. Ordering relations graph

elements of D. Two elements x and y in D are *consistent*, written $x \uparrow y$, iff they have a joint upper bound, i.e., $x \uparrow y \Leftrightarrow \exists\, z \in D : x \sqsubseteq z \wedge y \sqsubseteq z$; otherwise they are *inconsistent*. A subset X of D is *pairwise consistent*, written $X^{\Uparrow}$, iff every two elements in X are consistent in D, i.e., $X^{\Uparrow} \Leftrightarrow \forall x, y \in X : x \uparrow y$. The poset $(D, \sqsubseteq)$ is *coherent*, iff each pairwise consistent subset X of D has a least upper bound (lub) $\sqcup X$. An element $x \in D$ is a *complete prime*, iff for each subset X of D, which has a lub $\sqcup X$, holds that $x \sqsubseteq \sqcup X \Rightarrow \exists\, y \in X : x \sqsubseteq y$. Let $P = (D, \sqsubseteq)$ be a poset. We write $\mathfrak{P}_P$ for the set of complete primes of P. The poset $P = (D, \sqsubseteq)$ is *prime algebraic*, iff $\mathfrak{P}_P$ is denumerable and every element in D is the lub of the complete primes it dominates, i.e., $\forall\, x \in D : x = \sqcup\{y \mid y \in \mathfrak{P}_P \wedge y \sqsubseteq x\}$. A set S is *denumerable*, iff it is empty or there exists an enumeration of S that is a surjective mapping from the set of positive integers onto S. Figure 18 shows two prime-algebraic posets; their elements are sets (of events) ordered by set inclusion. The complete primes are written in bold typeface.

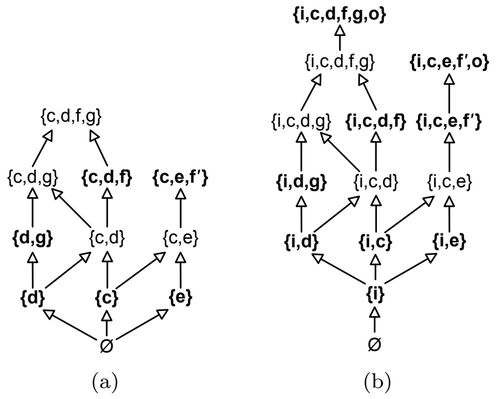


FIGURE 18. (a) Poset, and (b) augmented poset obtained from the graph in Figure 17

The behavior captured in an ordering relations graph can be translated to a prime algebraic partial order. Similar to [49], the elements of the partial order are the left-closed and conflict-free subsets of vertices of the graph. In our case the graph's vertices represent events and each such set of events describes the history of events of some run of a system. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be a graph and let W be a subset of V. W is *conflict-free*, iff $\forall\, v_1, v_2 \in W : (v_1, v_2) \notin A \vee (v_2, v_1) \notin A$. W is *left-closed*, iff $\forall\, v_1 \in W\ \forall\, v_2 \in V : (v_2, v_1) \in A \wedge (v_1, v_2) \notin A \Rightarrow v_2 \in W$. We define $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ as the partial order of left-closed and conflict-free subsets H of V, ordered by inclusion $\subseteq$. Figure 18(a) shows the poset $\mathcal{L}$ of the graph in Figure 17. Theorem 7.1, inspired by Theorem 8 in [49], characterizes the posets $\mathcal{L}[\mathcal{G}]$.

Theorem 7.1. Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph, let $B = \{a \in A \mid a^{-1} \notin A\}$. Then, $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ is a prime algebraic coherent partial order. Its complete primes are those elements of the form $[v] = \{v' \in V \mid (v', v) \in B^*\}$.

Proof. Let $X \subseteq H$ be pairwise consistent. Then, $\cup X$ is conflict-free. $\sqcup X = \cup X$ and, hence, $\mathcal{L}[\mathcal{G}]$ is coherent. Each $[v]$, $v \in V$, is clearly left-closed and conflict-free. Let $X \subseteq H$ have lub $\sqcup X$. X is pairwise consistent and $\sqcup X = \cup X$. Each $[v]$ is a complete prime. If $[v] \subseteq \cup X$, then $v \in \cup X$ and for some $x \in X$ holds $v \in x$ and, thus, $[v] \subseteq x$. It holds for each $X \in H$ that $X = \cup\{[v] \mid v \in \cup X\}$. Thus, each element of $\mathcal{L}[\mathcal{G}]$ is a lub of the complete primes below it. □

Given an ordering relations graph, one can construct $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ iteratively. Let h_1 and h_2 be subsets of V, such that $h_2 \setminus h_1 = \{v\}$. Then $h_1, h_2 \in H$, iff $h_1 = \varnothing$ or $\exists\, a \in h_1 : (v, a) \notin A$, and $\forall\, b \in h_1 : (b, v) \notin A \vee (v, b) \notin A$, and $\forall\, c \in V \setminus h_1, (c, v) \in A, (v, c) \notin A\ \exists\, d \in h_1 : (c, d), (d, c) \in A$.

Finally, we augment $\mathcal{L}[\mathcal{G}] = (H, \subseteq)$ with two fresh events $i, o \notin V$ to ensure the existence of a single source i and single sink o in the synthesis result. An *augmented* partial order of $\mathcal{G}$ is $\mathcal{L}^*[\mathcal{G}] = (H^*, \subseteq)$, where $H^* = \{\varnothing\} \cup \{h \cup \{i\} \mid h \in H\} \cup \{h \cup \{i, o\} \mid h \in H_{max}\}$. Figure 18(b) shows $\mathcal{L}^*$ of the graph in Figure 17. Adding new minimal and maximal elements i and o leaves the topology of the posets unchanged, so $\mathcal{L}^*[\mathcal{G}]$ is a prime algebraic coherent poset.

7.2. From partial orders to event structures

The step from an ordering relations graph $\mathcal{G}$ to its augmented poset $\mathcal{L}^*[\mathcal{G}]$ basically describes configurations of an unfolding that has the same ordering relations as $\mathcal{G}$. The complete primes of $\mathcal{L}^*[\mathcal{G}]$ play a special role: they identify single events. Synthesizing the unfolding itself requires to define these events and

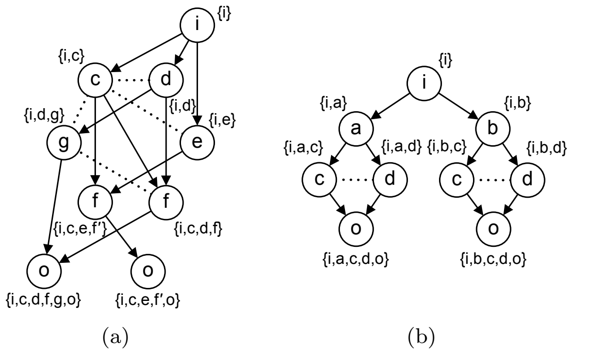


FIGURE 19. Event structures obtained from (a) Figure 18(b), and (b) the augmented poset of the graph in Figure 12(b)

their conflict relation explicitly. We do so by the help of the well-studied concept of an event structure [49].

Definition 7.1 (Labeled event structure).
An *event structure* is a triple $\mathcal{E} = (E, \leq, \oplus)$, where E is a set of events, $\leq$ is a partial order over E called the causality relation, and $\oplus$ is a symmetric and irreflexive relation in E, called the conflict relation that satisfies the principle of *conflict heredity*, i.e., $\forall e_1, e_2, e_3 \in E : e_1 \oplus e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \oplus e_3$. A *labeled* event structure $\mathcal{E} = (E, \leq, \oplus, \mathcal{C}, \kappa)$ additionally has a set $\mathcal{C}$ of *labels*, $\tau \in \mathcal{C}$, and $\kappa : E \to \mathcal{C}$ assigns to each event a label.

An ordering relations graph $\mathcal{G}$ differs from an event structure $\mathcal{E}$ in that $\mathcal{G}$ allows violations of conflict heredity. These violations, however, are not harmful; they express equivalent runs of a system. These equivalent runs are visible in the posets of Sect. 7.1 and become explicit in event structures. Each prime algebraic coherent poset $\mathcal{L}^*[\mathcal{G}] = (H, \subseteq)$ of an ordering relations graph $\mathcal{G}$ induces an event structure $\mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$: each complete prime becomes an event, the ordering relation $\subseteq$ induces the partial order $\leq$ over events, conflicts arise between events that have no joint least upper bound in $\mathcal{L}^*[\mathcal{G}]$. The resulting event structure can intuitively be understood as an unfolding of the ordering relations graph that adheres to conflict heredity. The formal definition is an extension of Def. 18 in [49]; it incorporates propagation of labels of an originative ordering relations graph to the corresponding event structure.

Definition 7.2 (Event structure of partial order).
Let $\mathcal{G} = (V, A, \mathcal{B}, \sigma)$ be an ordering relations graph and let $P = (H, \subseteq)$ be an (augmented) prime algebraic coherent partial order of $\mathcal{G}$. Then, $\mathcal{E}[P]$ is defined as the labeled event structure $(E, \leq, \oplus, \mathcal{C}, \kappa)$, where $E = \mathfrak{P}_P$, $\leq$ is $\subseteq$ restricted to $\mathfrak{P}_P$, for all $e_1, e_2 \in \mathfrak{P}_P : e_1 \oplus e_2$, iff e_1 and e_2 are inconsistent in P, and $\mathcal{C} = \mathcal{B} \cup \{\tau\}$. Let $e \in E$, and define $\hat{e}$ as $\hat{e} \in e \setminus \bigcup_{a \subset e, a \in H} a$. Then, $\kappa(e) = \sigma(\hat{e})$, if $\hat{e} \in V$; otherwise $\kappa(e) = \tau$, for all $e \in E$.

Figure 19(a) visualizes $\mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$ for the graph $\mathcal{G}$ of Figure 17. Events are complete primes of $\mathcal{L}^*[\mathcal{G}]$ (see in boldface in Figure 18 and next to vertices in Figure 19). Directed edges encode causality (transitive dependencies are not shown), dotted edges represent implicit concurrency, whereas an absence of an edge hints at a conflict relation. The particular example of the event structure in Figure 19(a) is structurally similar to the graph in Figure 17; they differ only in Figure 19(a) having additional fresh i, o events. In general, event structures tend to have a different structure compared to the originative graphs. For instance, Figure 19(b) shows the event structure derived from the augmented poset of the graph in Figure 12(b).

7.3. From event structures to occurrence nets

The event structure $\mathcal{E} = \mathcal{E}[\mathcal{L}^*[\mathcal{G}]]$ explicitly describes events of an unfolding that has the same ordering relations as the originative ordering relations graph $\mathcal{G}$. We obtain the unfolding by enriching $\mathcal{E}$ with conditions, that is, we translate $\mathcal{E}$ to an occurrence net. Nielsen et al. in [49] show a tight connection between event structures and occurrence nets. Let $N = (B, E, G)$ be an occurrence net. Then, $\xi[N] = (E, G^* \cap E^2, \#_N \cap E^2)$ is a corresponding event structure. The next theorem, borrowed from [49], defines the converse: how to construct an occurrence net from a given event structure.

Theorem 7.2. Let $\mathcal{E} = (E, \leq, \oplus)$, $E \neq \varnothing$, be an event structure. Then, there exists an occurrence net $\eta[\mathcal{E}]$, such that $\mathcal{E} = \xi[\eta[\mathcal{E}]]$.

Proof. Define the set $CE = \{x \subseteq E \mid \forall e_1, e_2 \in x : e_1 \neq e_2 \Rightarrow e_1 \# e_2\}$. The events of $\eta[\mathcal{E}]$ are exactly those in E. The set of conditions is defined by $B = \{\langle e, x\rangle \mid e \in E, x \in CE, \textit{ and } \forall e' \in x : e \leq e'\} \cup \{\langle 0, x\rangle \mid x \in CE, \textit{ and } x \neq \varnothing\}$. The flow relation is defined by $G = \{(\langle e, x\rangle, e') \mid \langle e, x\rangle \in B, e' \in x\} \cup \{(\langle 0, x\rangle, e') \mid \langle 0, x\rangle \in B, e' \in x\} \cup \{(e, \langle e, x\rangle) \mid \langle e, x\rangle \in B\}$. It follows, that $\eta[\mathcal{E}] = (B, E, G)$ is an occurrence net for which $\# = \oplus$, and hence $\xi[\eta[\mathcal{E}]] = \mathcal{E}$. $\square$

In the following we consider *labeled* event structures and correspondingly *labeled* occurrence nets to describe system behavior where a transition may occur in several contexts, e.g., after a join. A labeled occurrence net generalizes the notion of a branching process of a (labeled) net system (Definition 3.9) as it describes system behavior when the system is *not* known – which is the case here, as we want to synthesize one.

A labeled occurrence net $N = (B, E, G, \mathcal{T}, \lambda)$ is an occurrence net (B, E, G) (Definition 3.8) where $\lambda : B \cup E \to \mathcal{T}$, $\tau \in \mathcal{T}$, assigns each node $x \in B \cup E$ a label $\lambda(x)$. Every branching process $\beta = (N, \nu)$, $N = (B, E, G)$ of a labeled net system $S = (N_S, M_0)$, $N_S = (P, T, F, \mathcal{T}, \lambda)$ induces the labeled occurrence net $(B, E, G, \mathcal{T}, \nu \circ \lambda)$ that describes the ordering of occurrences of labels $\mathcal{T}$ instead of occurrences of transitions of S.

Theorem 7.2 canonically lifts to labeled event structures and labeled occurrence nets: each labeled event structure $\mathcal{E} = (E, \leq, \oplus, \mathcal{C}, \kappa)$ induces the labeled occurrence net $\eta[\mathcal{E}] = (B, E, G, \mathcal{C} \cup \{\tau\}, \kappa')$ which

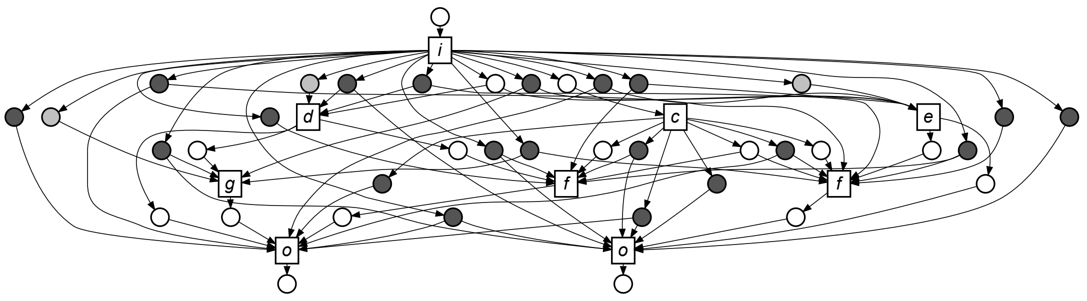


FIGURE 20. Occurrence net obtained from Figure 19(a) by Theorem 7.2 (without redundant conditions)

preserves the labels of events and assigns each condition label τ, i.e., for all $e \in E, \kappa'(e) = \kappa(e)$, and for all $b \in B, \kappa'(b) = \tau$.

Figure 20 shows the labeled occurrence net which is constructed from the event structure shown in Figure 19(a) using the principles of Theorem 7.2. Theorem 7.2 defines a "maximal" construction, see [49], i.e., the resulting nets tend to contain much redundancy. With Def. 7.3 we aim at preserving only essential behavioral dependencies.

DEFINITION 7.3 (Conditions).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net.

- A condition $b \in B$ is *redundant*, iff $b\bullet = \varnothing \wedge \exists\ b' \in B, b \neq b' : b' \in (\bullet b)\bullet$ or $\bullet b = \varnothing \wedge \exists\ b' \in B, b \neq b' : (b\bullet = b'\bullet) \wedge (\bullet b' \neq \varnothing)$.
- A condition $b \in B$ is *subsumed* by condition $b' \in B$, $b \neq b'$, iff $\bullet b = \bullet b' \wedge b\bullet \subseteq b'\bullet$.
- A condition $b \in B$ denotes a *transitive conflict* between events $e_1, e_2 \in E$, iff $\lambda(e_1) \neq \lambda(e_2)$ and there exists condition $b' \in B, b \neq b'$, and events $f_1, f_2 \in b'\bullet$, $(f_1 \rightsquigarrow_N e_1) \wedge (f_2 \rightsquigarrow_N e_2 \vee f_2 = e_2)$.
- Any other condition is *required*.

A redundant condition has no pre-event (post-event), and is not a pre-condition (post-condition) of the initial (a final) event. A subsumed condition b always has a sibling b' expressing the same constraints for larger set of events. A condition b denotes a transitive conflict between two events, if an "earlier" condition b' already denotes this conflict. Subsumed conditions are shaded light-grey in Figure 20 and transitive conflicts dark-grey. All these conditions can be removed from the occurrence net without loosing information about ordering of events.

An exception to transitive conflicts is a condition b shared by two post-events e_1, e_2 with the same label. Here, b not only expresses conflict, but also the *only* direct causal dependency of e_1 and e_2 on the pre-event of b; such a condition is required (Definition 7.3).

For synthesizing a Petri net from the occurrence net obtained from an ordering relations graph, we remove from the occurrence net first all redundant conditions, then all subsumed conditions, and finally all transitive conflicts. Removing these conditions from the net in Figure 20 yields the net in Figure 23. Note that all conditions are labeled τ. Condition b_9 highlights the exception to transitive conflicts: it is the only condition expressing that f depends on c and, hence, must be part of the occurrence net. Also the resulting net still contains a number of *implicit conditions*, i.e., conditions which could be removed from the net without changing the ordering relations such as b_{11} denoting $e_2 \rightsquigarrow e_8$ which is also expressed by the path $e_2, b_8, e_6, b_{15}, e_8$. We shall see that these remaining implicit conditions are vital for synthesizing a Petri net from a given occurrence net.

7.4. From occurrence nets to nets (the basic idea)

The occurrence net obtained by Theorem 7.2 and after removing redundant, subsumed, and transitive conflict conditions (Definition 7.3) is already a process model – though one with duplicate structures and multiple sinks. We obtain a more compact model with a single sink by *folding* the occurrence net.

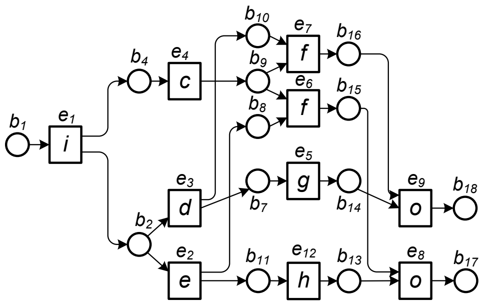


FIGURE 21. Occurrence net without implicit conditions.

We first illustrate the idea on a simple, special case and then present the general approach. Consider the occurrence net N in Figure 21 which contains *no* implicit conditions. Intuitively, we obtain a process model from N by folding any two nodes of N which have isomorphic successors into one node. This operation preserves all ordering relations and all behavior represented in N. The corresponding formal notion is an equivalence on the nodes of the occurrence net, called *future equivalence*, which is characterized co-inductively:

Two nodes of an occurrence net are future equivalent only if they have the same label and their post-sets are future equivalent.

The equivalence classes of the future equivalence define the nodes of the folded net: for each equivalence class, fold all its nodes into one node, preserving the arcs. The branching process of the folded net is exactly the original occurrence net, see [55, Theorem 8.7].

For the occurrence net N in Figure 21 consider the future equivalence $\sim_f$ with the classes $\{b_{17}, b_{18}\}$, $\{e_8, e_9\}$, $\{b_{15}, b_{16}\}$, $\{b_{13}, b_{14}\}$, $\{e_6, e_7\}$, $\{b_8, b_{10}\}$, and all other nodes remaining singleton. Folding N under $\sim_f$ yields the net in Figure 22 which has N as its branching process.

Each occurrence net has several future equivalences differing in how pre-conditions of events are folded. The algorithm for constructing a future equivalence is described next.

7.5. From occurrence nets to nets (the general case)

The general case of folding an occurrence net to a process model has one additional twist: the process model to be synthesized from the original ordering relations graph may have *unobservable* control flows between gateways *without* a task, e.g., the flow from x to y in Figure 2 is unobservable. Such flows can only be synthesized when the occurrence net to fold contains implicit conditions. The occurrence net obtained from Theorem 7.2 and reduced to its required conditions contains all implicit conditions having exactly one pre- and one post-event. These implicit conditions are an overapproximation of the unobservable control flow in the process model. Our folding procedure identifies during folding all implicit conditions that explain required unobservable control flow, and discards all others. Technically, we fold an occurrence net by first identifying a *folding equivalence* and then folding all nodes of an equivalence class into the same node of the process model. The folding equivalence has two properties: (1) it is a future equivalence, so that equivalent events of an occurrence net have equivalent successor events, and (2) pre-conditions and post-conditions of equivalent events have to be mutually equivalent. We first formalizing these notions, and then describe the procedure that finds such a folding equivalence for a given occurrence net.

Definition 7.4 (Future equivalence).

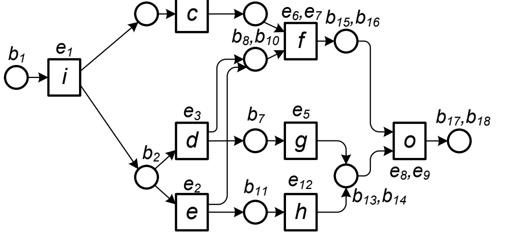


FIGURE 22. Folded net obtained from Figure 21

Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net. An event $e \in E$ is *direct successor* of an event $e' \in E$, written $e \in dSucc(e')$, iff $e' \rightsquigarrow_N e$ and for no e'' holds $e' \rightsquigarrow_N e'' \rightsquigarrow_N e$.

An equivalence relation $\sim \subseteq (B \times B) \cup (E \times E)$ is a *future equivalence* on N, iff the following properties hold:

- For all $x, y \in B \cup E$ holds, $x \sim y$ implies $\lambda(x) = \lambda(y)$ and $\neg(x \parallel_N y)$.
- For all $e, f \in E$ with $e \sim f$ holds: for each $e' \in dSucc(e)$ with $\lambda(e) = a$ exists $f' \in dSucc(f)$ with $\lambda(f) = a$ and $e' \sim f'$, and vice versa.

The future equivalence captures the essence of our behavioral equivalence criterion for structuring, fully concurrent bisimulation [6, Def. 12], in a stronger form that suits folding. Every future equivalence on an occurrence net N yields a fully concurrent bisimulation relation on the partially ordered runs described by N.[5] The converse does not hold as a future equivalence also considers invisible events of N. Merging future equivalent events of N into the same transition preserves the ordering relations encoded in N, see [55, Theorem 8.7] for the formal proof.

Folding an occurrence net N also requires to correctly fold pre- and post-conditions of events while treating implicit conditions in the right way. Thereby, the event with the largest number of non-implicit pre-conditions in an equivalence class determines the number of predecessors for the entire class; all other events in the class “fill up” their pre-set with implicit conditions; correspondingly for post-sets. To formalize this, we need to introduce some notions.

For an occurrence net $N = (B, E, G, \mathcal{T}, \lambda)$, let $implicit_N \subseteq B$ be the set of implicit conditions of N, i.e., $b \in implicit_N$ iff $\{e_1\} = \bullet b, b\bullet = \{e_2\}$ and there is a path from e_1 to e_2 in N without b.

Let $\sim \subseteq (B \times B) \cup (E \times E)$ be an equivalence relation. For an equivalence class $\langle x \rangle$ of $\sim$, let $maxpre(\langle x \rangle)$ be the largest number of non-implicit predecessors of all members of $\langle x \rangle$, i.e., $maxpre(\langle x \rangle) = |\bullet x'|, x' \in \langle x \rangle$ s.t. for all $x'' \in \langle x \rangle$ holds $|\bullet x' \setminus implicit_N| \geq |\bullet x'' \setminus implicit_N|$; correspondingly let $maxpost(\langle x \rangle)$ be the largest number of non-implicit successors of all members of $\langle x \rangle$. For sets $X, Y \subseteq B$ we write $X \sim Y$ iff $X = \{x_1, \ldots, x_k\}, Y = \{y_1, \ldots, y_k\}$ and $x_i \sim y_i$, for all $1 \leq i \leq k$.

Definition 7.5 (Folding equivalence).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net, $\forall\ b \in B : \lambda(b) = \tau$. An equivalence relation $\sim \subseteq (B \times B) \cup (E \times E)$ *preserves the environment of events* iff for each equivalence class $\langle e \rangle$ of $\sim$, $e \in E$, and for all $e_1, e_2 \in \langle e \rangle$ holds:

- $\exists B_1 \subseteq \bullet e_1, B_2 \subseteq \bullet e_2 : B_1 \sim B_2 \wedge \forall i \in \{1, 2\} : |B_i| = maxpre(\langle e \rangle) \wedge \bullet e_i \setminus B_i \subseteq implicit_N$, and
- $\exists B_1 \subseteq e_1\bullet, B_2 \subseteq e_2\bullet : B_1 \sim B_2 \wedge \forall i \in \{1, 2\} : |B_i| = maxpost(\langle e \rangle) \wedge e_i \bullet \setminus B_i \subseteq implicit_N$.

[5]Every prefix π of a labeled occurrence net N that contains no conflicting events is a partially ordered run [46].

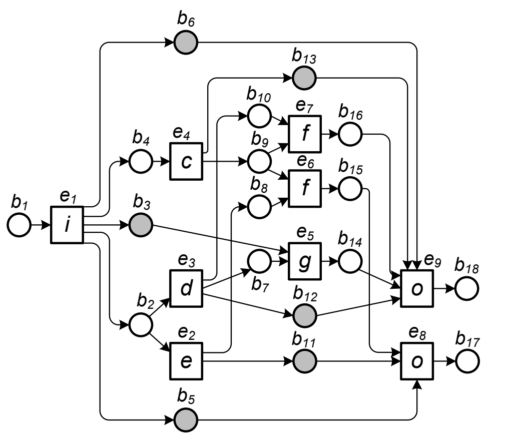


FIGURE 23. Simplified occurrence net obtained from Figure 20

An equivalence $\sim_f \subseteq (B \times B) \cup (E \times E)$ is a *folding equivalence* on N iff $\sim_f$ is a future equivalence and preserves the environment of events.

Considering the occurrence net N in Figure 23, the equivalence $\sim_f$ with the classes $\{b_{17}, b_{18}\}$, $\{e_8, e_9\}$, $\{b_{11}, b_{14}\}$, $\{b_{15}, b_{16}\}$, $\{e_6, e_7\}$, $\{b_8, b_{10}\}$, and all other nodes remaining singleton, is a folding equivalence on N. In particular, in the equivalence class $\{e_8, e_9\}$ the pre-set of event e_9 defines that all events have to have 2 pre-conditions that are mutually equivalent. Hence, $\{b_{15}, b_{16}\}$ and $\{b_{11}, b_{14}\}$ are equivalence classes of $\sim_f$ where the second class contains the implicit condition b_{14}. The remaining pre-conditions of $\{e_8, e_9\}$ need not be equivalent.

Without implicit conditions, folding would not succeed as it can be seen from the occurrence net in Figure 23 where implicit conditions are highlighted grey. The two equivalent events e_8 and e_9 differ in the number of non-implicit pre-conditions. To consistently fold e_8 and e_9 w.r.t. their pre- and post-sets, the pre-set of e_8 has to be extended with one implicit condition during folding.

We fold an occurrence net to a Petri net by merging all nodes of a folding equivalence class into the same node – except for equivalence classes consisting of implicit conditions only, these are discarded.

DEFINITION 7.6 (Folded net).
Let $N = (B, E, G, \mathcal{T}, \lambda)$ be a labeled occurrence net s.t. $\forall\ b \in B : \lambda(b) = \tau$. Let $\sim_f$ be a folding equivalence on N; write $\langle x \rangle_f = \{y \mid y \sim_f x\}$ for the equivalence class of x. The *folded net of N under* $\sim_f$ is the net $N_f = (B_f, E_f, G_f, \mathcal{T}, \lambda_f)$ where

- $B_f = \{\langle b \rangle_f \mid b \in B, \langle b \rangle_f \notin \mathit{implicit}_N\}$,
- $E_f = \{\langle e \rangle_f \mid e \in E\}$,
- $G_f = \{(\langle x \rangle_f, \langle y \rangle_f) \mid (x, y) \in G, \langle x \rangle_f, \langle y \rangle_f \in B_f \cup E_f\}$, and
- $\lambda_f(\langle x \rangle_f) = \lambda(x)$.

Folding N of Figure 23 under $\sim_f$ given above yields the net N_f shown in Figure 24. Folding N into N_f preserves the behavior of N, see [55, Theorem 8.7].

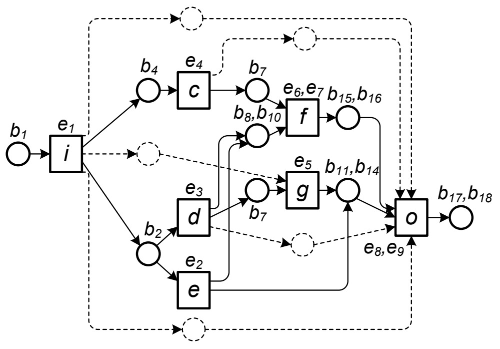


FIGURE 24. Folded net obtained from Figure 23

A simple algorithm to compute a folding equivalence traverses the given finite occurrence net backwards in a breadth-first manner. The conditions of the occurrence net without any post-event have equivalent futures. Correspondingly, their pre-events with the same label have equivalent futures. Building the folding equivalence backwards in this way ensures that only future equivalent events are put into the same equivalence class. Branching and backtracking are used whenever for a condition b there are two or more pairwise concurrent conditions that could be folded with b. Each option is explored and the most-compact folding is chosen. For instance in Figure 21 after folding $b_{17} \sim_f b_{18}$ and $e_8 \sim_f e_9$, for b_{15} the folding options b_{14} and b_{16} can be explored; backtracking yields b_{16} as the better match for b_{15} because of their f-labeled pre-events.

When extending the folding equivalence to pre-conditions of equivalent events E', for instance for $E' = \{e_8, e_9\}$, implicit conditions are taken into account as follows:

- Pick the event $e \in E'$ with the largest set B' of non-implicit pre-conditions, e.g., $\{b_{14}, b_{16}\} \subset \bullet e_9$.
- For each $b \in B'$, extend the folding equivalence with a non-implicit or implicit condition $b' \in \bullet e'$, for all other events $e' \in E'$, preferring non-implicit conditions over implicit ones, e.g., $b_{16} \sim_f b_{15}$, $b_{14} \sim_f b_{11}$.
- Finally, remove all non-implicit conditions not required in this step, e.g., b_5, b_6, b_{13}.

The second step ensures that pre-sets of equivalent events are preserved, the third step ensures that post-sets of equivalent events (that are identified in subsequent steps) are preserved. Various heuristics improve exploration and backtracking when matching pre-conditions of events with each other. When the future equivalence cannot be extended further, the net can be folded according to Def. 7.6. Applying this procedure on our example yields the folded net shown in Figure 24 without the dashed conditions and arcs.

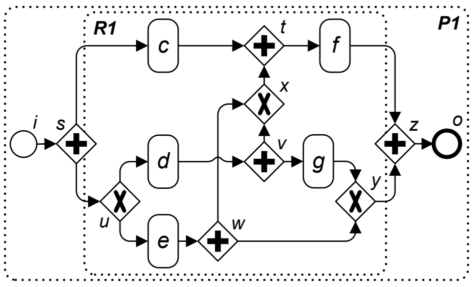


FIGURE 25. Process model obtained from Figure 24

7.6. From nets to process models

The folding was the second to last step in synthesizing a process model from a given ordering relations graph. We obtained a Petri net N_f which we now transform into a process model P using the principles of Figure 5 in the reverse direction.

The initial transition i (final transition o) is mapped to the source (sink) node of P. Every other transition of N_f becomes a task of P. Gateways of P follow from non-singleton pre- and postsets of nodes of N_f. A transition t with two or more pre-places is preceded by an *and* join; two or more post-places of t define an *and* split; the pre- and postsets of places define *xor* splits and joins, respectively; *and* gateways are always positioned closer to the task. In our example, $e_1\bullet$ defines *and* split s in Figure 25, $b_2\bullet$ defines *xor* split u, $e_3\bullet$ defines *and* split v, $\bullet\langle e_6, e_7\rangle$ defines *and* join t, and $\bullet\langle b_6, b_8\rangle$ defines *xor* join x positioned between t and v (*and* gateways closer to tasks); correspondingly for all other gateways. The arc from e_2 to $\langle b_{11}, b_{14}\rangle$ which was obtained from a transitive conflict (Definition 7.3) results in an important control-flow arc from w to y without any task.

This concludes our technique for maximal structuring. Returning to our running example, we complete the maximal structuring of the model of Figure 2(a) by placing the synthesized process model $R1$ of Figure 25 at the corresponding spot in the RPST of Figure 14(bottom). Recall that this RPST was obtained from the original process model using Algorithm 1. Placing $R1$ at its designated spot yields the maximally-structured model of P shown in Figure 2(b).

8. EVALUATION

The overall approach has been implemented in a tool, namely `bpstruct`, which is publicly available[6]. Using `bpstruct` we conducted an evaluation to assess the performance of our techniques and to analyze the amount of duplication introduced during the structuring. All tests were performed on a laptop with a dual core Intel processor, 2.53 GHz, 4 GB of memory, running Microsoft Vista and SUN Java Virtual Machine version 1.6 (with 512 MB of allocated memory). To eliminate load time from the measures, each test was executed five times, and we recorded the average execution time of the second to fifth run. In the following, we provide details about the dataset used for the study and discuss the results of the experiments.

8.1. Dataset

The study was conducted on a collection of process models extracted from industrial practice that has been publicly released for research purposes [56]. More precisely, we used the set of WF-nets provided in `Woflan` [57, 58] file format[7]. In contrast to the original collection, where a large number of process models has multiple start nodes and/or multiple end nodes, the WF-nets have been completed such that each WF-net has a single source and a single sink place while preserving the original behavior. The reader is referred to [56] for a detailed description of the dataset and the underlying completion process.

In a first stage we removed all unsound WF-nets from the collection. Every sound WF-net was parsed using the RPST decomposition as described in [59]. Every bond and polygon in the RPST was abstracted to a single transition. For each rigid component identified in the RPST a process model was synthesized using the approach described in Section 7.6. Two nodes were added to mark the entry and exit points, respectively. This procedure yielded a total of 170 sound unstructured process models. Among them, 115 are *heterogeneous* acyclic rigids (acyclic rigids with *xor* and *and* gateways), 39 are *and* rigids (acyclic rigids with *and* gateways only), 14 are cyclic rigids and 2 are *xor* rigids (rigids with *xor* gateways only). For the purpose of this study, we only kept models with heterogeneous acylic rigids and *and* rigids (154 process models). To ease the analysis, we further classified the heterogeneous rigids into "structurable" and "maximally structurable", accordingly. Table 2 summarizes the size of the models used for the study. In the table, *and* rigids are referred to as *inherently unstructured*. The size of the models after structuring is shown in parenthesis. There were two exceptionally large process models in the dataset, one "structurable" and the other one "maximally structurable". Both models have been excluded when calculating the average sizes. However, their sizes correspond to maximum values in their structural classes in Table 2.

8.2. Results

Our study reveals that out of the 154 process models, 110 models (71.43%) can be structured into well-structured models. More importantly, there exist models, in our case 5 (3.25%), which require the extension proposed in this article to achieve maximal-structuring, while 39 models (25.32%) are inherently unstructured. Concerning performance, Figure 26(a) presents the

[6]http://code.google.com/p/bpstruct

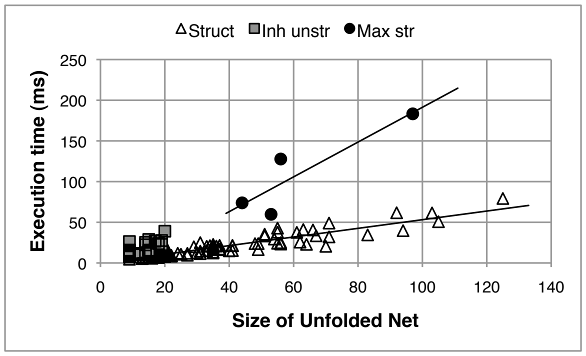
[7]http://service-technology.org/soundness

TABLE 2. Structural information on process models in the dataset

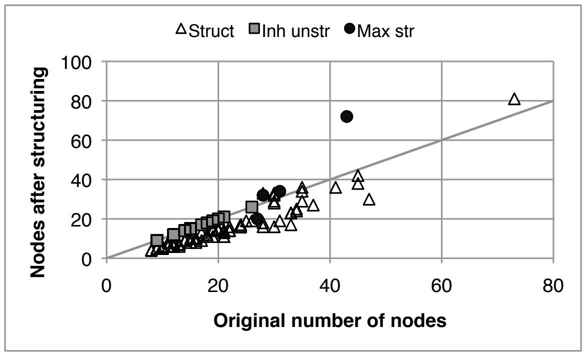
	Structurable	**Maximally structurable**	**Inherently unstructured**
Number of models	110	5	39
Avg number of nodes/arcs	21/31 (14/16)	32/51 (40/47)	12/14
Max number of nodes/arcs	119/195 (1178/1346)	124/173 (545/631)	26/32

execution times relative to the number of events in the proper complete prefix unfolding, including a trend line for each structural class. It can be easily noted that the execution time is highly correlated to the size of the proper prefix. We observed a significant difference in the processing time for "maximally structurable" models compared to "structurable" models. Nevertheless, the average time for structuring was in the order of milliseconds, with a few exceptions. We found two exceptionally large cases: one model ("structurable") with 119 nodes (the proper prefix had 2429 events) and the other one ("maximally structurable") with 124 nodes (the proper prefix had 913 events), requiring 9 and 1 seconds for structuring, respectively.

Figure 26(b) presents the variation in the size of models, i.e., number of nodes, after structuring. On average, the model's size decreased by about 15%, with a standard deviation of 0.817. This can be easily confirmed in the figure, as most of the points are located under the diagonal (which corresponds to a ratio 1:1), particularly in the case of "structurable" models. Recall that models were, in most of the cases, augmented with some additional elements (i.e., *and* gateways and transitive flow relations) to transform them into single exit models. Therefore, the reduction in size can be explained by the removal of gateways and transitive flow during the structuring perfomed by `bpstruct`. Conversely, the sizes of *and* rigids remained the same for this dataset. In additional internal experiments, we found that for certain *and* rigids it is also possible to observe a reduction on the size of the (maximally-)structured version when redundant elements are eliminated, e.g., spurious flow relations/gateways. The dataset used in our experiments is publicly available at `bpstruct` web site[8], including the graphical version of the process models in PDF file format.



(a) Average structuring time



(b) Ratio of duplication

FIGURE 26. Experimental results

9. CONCLUSION

In [6], we presented for the first time the full characterization of the class of acyclic process models that have an equivalent structured version along with a structuring method. The method stops when the input model contains an inherently unstructured process component. This article completes the approach by providing a method to synthesize the components corresponding to inherently unstructured parts of the input model.

Close to our setting, the problem of synthesizing nets from behavioral specifications has been a line of active research for about two decades [60, 61]. This area has given rise to a rich body of knowledge and to a number of tools, in particular `petrify` [60] and `viptool` [61]. Yet, these solutions fail in our setting: `petrify` aims at maximizing concurrency while our synthesis preserves given concurrency, `viptool` synthesizes nets with arc weights, which do not map to process models.

The approach is implemented in a tool, namely `bpstruct`, which is publicly available. The running time of our structuring technique is mostly dominated by the time required to compute proper prefixes, which for safe systems is $O((|B|/n)^n)$ [50], where B is the set of conditions of the prefix and n is the maximal size of the presets of the transitions in the originative system. All other steps can be accomplished in linear time. Concerning the extension for maximal structuring, the theoretic discussion in this article implies exponential time and space complexities when constructing posets (this is due to our intent to stay close to the existing theory). However, in practice, given an ordering relations graph one can construct a poset which only contains information from the graph, without introducing duplicate events, and thus stay linear to the size of the graph. At the theoretical

[8]http://code.google.com/p/bpstruct/wiki/MaxStructEvaluation

level this requires introduction of a concept of a cutoff for posets followed by an adjustment of the theories along subsequent transformation steps. The folding step reverses the unfolding and, thus, in the best case can be performed in the same time.

The fact that the running time of structuring depends on the size of the result, allows to introduce heuristics to terminate computation when the result gets large, e.g., when the event duplication factor is larger than two. Moreover, we envision a technique which can decide on-line, i.e., during the construction of the proper prefix, that from now on the prefix defines an ordering relations graph which contains a primitive module and, thus, the model cannot be structured. However, in practice we have never observed such a need with our implementation in most cases delivering results in milliseconds. Note that the amount of task duplication in the structured models is controlled by proper prefixes. As proper prefixes are always minimal, see the discussion in Section 5.1, the duplication introduced by our technique is minimal.

The employed notion of behavioral equivalence, i.e., fully concurrent bisimulation, cares only about the opportunity to extend equivalent runs of process models with the same task (observable transition) possibly by skipping several gateways (silent transitions), see [26]. Technically, this implies that our structuring algorithm might suppress implicit decision points (an *xor* split followed by another gateway) of an unstructured process model in its structured version. To avoid an aggregation of implicit decisions into a single decision, one can materialize them, i.e., introduce an observable *decision task* which follows the *xor* split. Decision tasks should be treated as all other observable tasks during structuring. Afterwards, they can be converted back into the decision points in the resulting structured model.

It is worth pointing out that spurious control flow relations in unstructured process models may forbid their structuring. However, these relations can be suppressed in the preprocessing step by unfolding and then refolding the model, by following the principles described in Sections 5 and 7. The refolded model is (in some sense) in a canonical form and should be employed for structuring; note that the refolded model is sometimes well-structured. The detailed analysis of such a preprocessing step is left for future work.

Our ongoing work aims at extending the method to handle models with loops. In our initial investigations, we try to derive a condition for unfolding truncation which results in prefixes with sufficient information to recognize all SESE loop components hidden within process models. The first results show that such a condition must be evaluated iteratively as complex overlapping loop topologies keep unfolding.

REFERENCES

[1] Weske, M. (2007) *Business Process Management: Concepts, Languages, Architectures.* Springer Verlag.

[2] van der Aalst, W. M. P. and Stahl, C. (2011) *Modeling Business Processes: A Petri Net-Oriented Approach.* Cambridge MA: MIT Press.

[3] Dumas, M., van der Aalst, W. M. P., and ter Hofstede, A. H. M. (2005) *Process-Aware Information Systems: Bridging People and Software through Process Technology.* Wiley-Interscience, Hoboken, NJ.

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

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

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

[7] Business process model and notation (BPMN) version 2.0. http://www.omg.org/spec/BPMN/2.0/PDF/.

[8] Keller, G., Nüttgens, M., and Scheer, A.-W. (1992) Semantische Prozeßmodellierung auf der Grundlage 'Ereignisgesteuerter Prozeßketten (EPK)'. Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi). Universität des Saarlandes. In German.

[9] Kitzmann, I., König, C., Lübke, D., and Singer, L. (2009) A simple algorithm for automatic layout of BPMN processes. *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems (CES)*, pp. 391–398. IEEE Computer Society.

[10] Effinger, P., Siebenhaller, M., and Kaufmann, M. (2009) An interactive layout tool for BPMN. *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems (CES)*, pp. 399–406. IEEE Computer Society.

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

[12] Laguna, M. and Marklund, J. (2005) *Business Process Modeling, Simulation, and Design.* Prentice Hall.

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

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

[15] Weidlich, M., Decker, G., Großkopf, A., and Weske, M. (2008) BPEL to BPMN: The myth of a straight-forward mapping. *OTM Conferences*, Lecture Notes in Computer Science, **5331**, pp. 265–282. Springer.

[16] Mazanek, S. and Hanus, M. (2011) Constructing a bidirectional transformation between BPMN and BPEL with a functional logic programming language. *Journal of Visual Languages and Computing (VLC)*, **22**, 66–89.

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

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

[19] Uba, R., Dumas, M., García-Bañuelos, L., and Rosa, M. L. (2011) Clone detection in repositories of business process models. *Business Process Management (BPM)* Lecture Notes in Computer Science. Springer.

[20] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987) The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **9**, 319–349.

[21] (2007). Web services business process execution language version 2.0. committee specification. http://www.oasis-open.org/committees/download.php/22475/wsbpel-v2.0-CS01.pdf.

[22] Reichert, M. and Dadam, P. (1998) ADEPT$_{\text{flex}}$-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems (JIIS)*, **10**, 93–129.

[23] Rinderle, S., Reichert, M., and Dadam, P. (2004) Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases (DPD)*, **16**, 91–116.

[24] van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. J. M. M. (2003) Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering (DKE)*, **47**, 237–267.

[25] Elliger, F., Polyvyanyy, A., and Weske, M. (2010) On separation of concurrency and conflicts in acyclic process models. *Enterprise Modelling and Information Systems Architectures (EMISA)*, Lecture Notes in Informatics, **172**, pp. 25–36. GI.

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

[27] Dijkstra, E. W. (1968) Letters to the editor: Go To statement considered harmful. *Communications of the ACM (CACM)*, **11**, 147–148.

[28] Hopkins, M. E. (1972) A case for the GOTO. *ACM Annual Conference*, pp. 787–790. ACM.

[29] Wulf, W. A. (1972) A case against the GOTO. *ACM Annual Conference*, pp. 791–797. ACM.

[30] Rubin, F. (1987) “GOTO considered harmful” considered harmful. *Communications of the ACM (CACM)*, **30**, 195–196.

[31] Moore, D., Musciano, C., Liebhaber, M. J., Lott, S. F., and Starr, L. (1987) ““GOTO considered harmful” considered harmful” considered harmful. *Communications of the ACM (CACM)*, **30**, 351—-355.

[32] Williams, M. H. (1977) Generating structured flow diagrams: The nature of unstructuredness. *The Computer Journal (CJ)*, **20**, 45–50.

[33] Williams, M. H. and Ossher, H. L. (1978) Conversion of unstructured flow diagrams to structured form. *The Computer Journal (CJ)*, **21**, 161–167.

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

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

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

[37] Koehler, J. and Hauser, R. (2004) Untangling unstructured cyclic flows – a solution based on continuations. *CoopIS/DOA/ODBASE*, Lecture Notes in Computer Science, **3290**, pp. 121–138. Springer.

[38] Lohmann, N. and Kleine, J. (2008) Fully-automatic translation of open workflow net models into simple abstract BPEL processes. *Modellierung*, LNI, **127**, pp. 57–72. GI.

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

[40] Polyvyanyy, A., García-Bañuelos, L., and Weske, M. (2009) Unveiling hidden unstructured regions in process models. *OTM Conferences*, Lecture Notes in Computer Science, **5870**, pp. 340–356. Springer.

[41] Best, E. and Shields, M. W. (1983) Some equivalence results for free choice nets and simple nets and on the periodicity of live free choice nets. *Colloquium on Trees in Algebra and Programming (CAAP)*, Lecture Notes in Computer Science, **159**, pp. 141–154. Springer.

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

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

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

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

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

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

[48] McMillan, K. L. (1995) A technique of state space search based on unfolding. *Formal Methods in System Design (FMSD)*, **6**, 45–65.

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

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

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

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

[53] van Hee, K. M., Sidorova, N., and Voorhoeve, M. (2003) Soundness and separability of workflow nets in the stepwise refinement approach. *Applications and Theory of Petri Nets (ICATPN/APN)*, Lecture Notes in Computer Science, **2679**, pp. 337–356. Springer.

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

[55] Fahland, D. (2010) From Scenarios To Components. PhD thesis Humboldt-Universität zu Berlin and Technische Universiteit Eindhoven.

[56] Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., and Wolf, K. (2011) Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, **70**, 448–466.

[57] Verbeek, H. M. W. E. and van der Aalst, W. M. P. (2000) Woflan 2.0: A Petri-net-based workflow diagnosis tool. *Applications and Theory of Petri Nets (ICATPN/APN)*, pp. 475–484.

[58] Verbeek, H. M. W. E., Basten, T., and van der Aalst, W. M. P. (2001) Diagnosing workflow processes using woflan. *The Computer Journal (CJ)*, **44**, 246–279.

[59] Polyvyanyy, A., Weidlich, M., and Weske, M. (2011) Connectivity of workflow nets: The foundations of stepwise verification. *Acta Informatica (ACTA)*, **48**, 213–242.

[60] Cortadella, J., Kishinevsky, M., Lavagno, L., and Yakovlev, A. (1998) Deriving Petri nets for finite transition systems. *IEEE Transactions on Computers (TC)*, **47**, 859–882.

[61] Bergenthum, R., Desel, J., and Mauser, S. (2009) Comparison of different algorithms to synthesize a Petri net from a partial language. *Transactions on Petri Nets and Other Models of Concurrency (TOPNOC)*, **3**, 216–243.