Automatic Generation of Optimized Process Models from Declarative Specifications

Richard Mrasek(✉), Jutta Mülle, and Klemens Böhm

Institute for Program Structures and Data Organization,
Karlsruhe Institute of Technology (KIT), Karlsruhe 76131, Germany
{richard.mrasek,jutta.muelle,klemens.boehm}@kit.edu

Abstract. Process models often are generic, i. e., describe similar cases or contexts. For instance, a process model for commissioning can cover both vehicles with an automatic and with a manual transmission, by executing alternative tasks. A generic process model is not optimal compared to one tailored to a specific context. Given a declarative specification of the constraints and a specific context, we study how to automatically generate a good process model and propose a novel approach. We focus on the restricted case that there are not any repetitions of a task, as is the case in commissioning and elsewhere, e. g., manufacturing. Our approach uses a probabilistic search to find a good process model according to quality criteria. It can handle complex real-world specifications containing several hundred constraints and more than one hundred tasks. The process models generated with our scheme are superior (nearly twice as fast) to ones designed by professional modelers by hand.

Keywords: Process synthesis · Automatic process generation · Commissioning processes · Business process modeling

1 Introduction

Scheduling tasks so that the overall execution is efficient and at the same time no constraints are violated continues to be a fundamentally important problem. Process models describe the possible arrangements of the tasks.

***Example* 1.** *Our application scenario is commissioning. Commissioning means configuring and testing the electronic components of a vehicle during its production. Process models describe the arrangement of the configuration and testing tasks. For instance, a factory worker has to configure the transmission and to activate the anti-theft system. The transmission can either be manual, i. e., Task* M *does the configuration, or automatic (Task* A*). Task* T *activates the anti-theft system. Before the activation, a central computer needs to generate a master key (Task* G*), and it opens the connection to the specific control unit (Task* O*). The connection has to be closed before the process finishes (Task* C*). The configuration of the transmission and the activation of the anti-theft system require a running engine; Task* E *turns it on. Figure 1(a) shows the tasks that may be* a)

	Task	proc. time
E	: Start **E**ngine	1s
M	: Conf. **M**anual transmission	5s
A	: Conf. **A**utomatic transmission	2s
T	: Activate anti-**T**heft system	1s
C	: **C**lose Connection	1s
O	: **O**pen Connection	1s
G	: **G**enerate Master Key	5s

b)

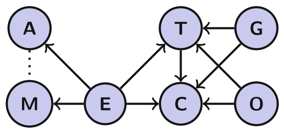


Fig. 1. The Tasks for Commissioning (a) and the Ordering Relationship Graph (b)

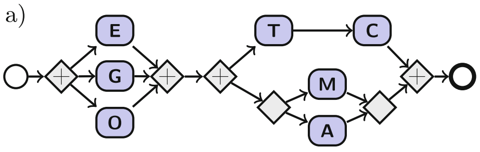
part of the commissioning. The second column is the expected processing time. Commissioning always has a context, i.e., the variation of the vehicle, its components, their relationships and the constraints the vehicle currently tested must fulfill. The variation determines which tasks have to be executed, e. g., a car with a manual transmission requires different tasks than a car with an automatic one.

A context c determines the tasks $\mathcal{T}_c$ required for a process. It is infeasible to model all processes for each possible set of required tasks by hand. This calls for generic process models for several contexts. With such generic models however, one optimal arrangement of tasks for any context does not exist.

Example 2. *The context characteristic* transmission *determines the required tasks as follows: If the vehicle has an automatic transmission, the commissioning requires execution of the tasks* $\mathcal{T}_c = \{\mathsf{E}, \mathsf{A}, \mathsf{T}, \mathsf{C}, \mathsf{O}, \mathsf{G}\}$*; for a manual transmission in turn the tasks are* $\mathcal{T}_c = \{\mathsf{E}, \mathsf{M}, \mathsf{T}, \mathsf{C}, \mathsf{O}, \mathsf{G}\}$*. Figure 1(b) shows the dependencies between the tasks as a graph. Directed edges represent ordering dependencies, while dashed lines represent exclusive dependencies. The graph is the declarative specification we generate the process model from. The extended version of this paper [17] shows how one can generate such a specification from input in other languages. Section 2 will introduce the notation behind that graph structure.*

Figure 2 shows two generic process models that comply with the dependency graph of Figure 1(b). Figure 2(a) has a shorter processing time if the transmission is automatic (7s to 10s). For a manual transmission in turn, the process model in Figure 2(b) has a shorter processing time (8s to 10s).

a)



b)

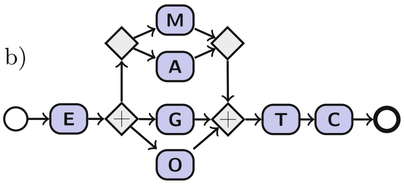


Fig. 2. Two Distinct Process Models for the Graph in Figure 1(b)

With at least one process model for each possible context, the number of such models increases exponentially with the number of context characteristics. For instance, 10 context characteristics that are Boolean in nature result in 1024 process models. Next, several models typically are possible for a given context. The problem studied here is how to generate a good process model for a given context from a declarative specification. The model should be good according to predefined quality criteria, e. g., throughput time. Process models that comply with the dependencies can be very different with respect to quality and performance criteria. Section 4 will show that models generated with our approach are about 50% faster than ones designed by professionals with years of experience. We focus on the restricted case that there are no repetitions, i. e., on process trees with inner nodes SEQ, AND, XOR, but not LOOP. There is a number of settings with this characteristic, for instance in manufacturing. In particular, loops are unnatural in commissioning processes, since a feature is tested only once. On the other hand, if a problem occurs and is fixed, a new commissioning process is started. Another assumption, which also holds for commissioning and elsewhere, is that context information together with experience from the past allows to reliably estimate the processing time of individual tasks.

The generation of a process model from a declarative specification bears several challenges. There often is a great variety of models that fulfill the specification, as mentioned before. To illustrate, the sequential arrangement of n nodes, in the absence of any constraint, give way to $n!$ different process models. For the largest process in our evaluation, 4.12×10^{340} models are possible. Generating all possible models is not possible. It is challenging to detect a good process model that does not violate any constraint.

***Example* 3.** *There are four tasks A, B, C, and D. Suppose that the following constraints exist: B must always occur before A and C ($B \rightarrow A$, $B \rightarrow C$), and D always occurs before C ($D \rightarrow C$). It seems to be a good idea to put A and C in parallel, because this might reduce the throughput time. But putting A and C in parallel rules out having A and D in parallel.*

Related work in process synthesis is fully automatic only for processes that are fully specified by their dependencies [6][25]. In case of an under-specification, [6] requires a process modeler to manually make decisions, and [25] requires a manual clustering of the constraints. This is not practical, because of the daunting number of possible models. To this end, we propose a novel process synthesis algorithm whose output on the one hand complies with the dependencies and on the other hand is good according to predefined criteria. Our approach is as follows: First, it uses a modular decomposition of the dependencies to detect the fully specified regions of the process as well as the under-specified ones, so-called prime components. For each prime component, our approach partitions the corresponding ordering graph systematically, as follows. It selects a pivot element and generates several smaller ordering graphs from the pivot partition. We reduce the problem in a divide and conquer fashion until it is small enough to explicitly generate all possible models. We repeat this for different pivot elements to have a better coverage according to our quality criterion the throughput time of the process. Other criteria such as overall energy consumption are possible as well. As we show in the evaluation with thousands of non-trivial process models, our approach is efficient, i. e., is able to test thousands of models in under a second, checking for complex constraints. On average, our approach nearly halves the processing time compared to the reference processes, which already are the output of a careful intellectual design. Our approach can handle complex real-world specifications containing several hundred dependencies as well as more than one hundred tasks. In our evaluation, the process models generated contain between 98 and 185 tasks, and their arrangement typically is nontrivial.

Section 2 introduces some fundamentals. Section 3 describes our algorithm for the process generation. Section 4 features our evaluation. Section 5 discusses related work, and Section 6 concludes.

2 Fundamentals

A meaningful input for process synthesis is the declarative specification in the form of an ordering relation graph (ORG) [21]. The modular decomposition of a graph yields its components and implies a hierarchical structure of components called the Modular Decomposition Tree (MDT)[15], see Subsection 2.3. The MDT separates the under-specified regions from the fully specified ones.

2.1 Ordering Relation Graph

In an ordering relation graph, each node represents a task. Each edge represents a dependency between tasks. The dependencies consist of ordering dependencies, i. e., in which order do the tasks occur, and exclusive dependencies, i. e., when do two tasks exclude each other.

Definition 1. *The ordering relation graph is a directed attributed graph $G = (V, E)$, with V being nodes and $E \subseteq V \times V$ the edges. Each node corresponds to a task. E consists of two subsets $E_\rightarrow$ and $E_\#$ such that $E = E_\rightarrow \cup E_\#$ and $E_\rightarrow \cap E_\# = \emptyset$. $E_\rightarrow$ defines the ordering relation, i. e., two tasks that should be in a specific order have an edge in $E_\rightarrow$. $E_\rightarrow$ is transitive and anti-symmetric:*

$$\begin{aligned} (\textit{transitive}) \quad & \forall (x,y),(y,z) \in E_\rightarrow : (x,z) \in E_\rightarrow \\ (\textit{antisymmetric}) \quad & \forall (x,y) \in E_\rightarrow : (y,x) \notin E_\rightarrow \end{aligned}$$

$E_\#$ defines the exclusiveness relation, i. e., if two tasks exclude each other they share an edge in $E_\#$. $E_\#$ is symmetric, i. e., $\forall (x,y) \in E_\# : (y,x) \in E_\#$. We do not allow self-edges, i. e., $\forall v \in V : (v,v) \notin E$.

Note that $E_\rightarrow$ does not contain any cycle. For each task we determine the processing time. The average error of the estimated execution times of our tasks from our application scenario is less than 17%. We had calculated these times by analyzing the logs of existing traces.

Definition 2. *The neighborhoods $N^{out}(v)$, $N^{in}(v)$ of a node v are defined as:*

$$N^{out}(v) := \{w \mid w \in V \wedge (v,w) \in E_{\rightarrow}\} \quad N^{in}(v) := \{w \mid w \in V \wedge (w,v) \in E_{\rightarrow}\}$$

$N^{out}(v)$ is the set of nodes with an incoming ordering edge from v. $N^{in}(v)$ is the set of nodes that have an outgoing ordering edge to v. For a set of nodes V the incoming and outgoing set are defined as $N^{out}(V) := \bigcup_{v \in V} N^{out}(v)$ and $N^{in}(V) := \bigcup_{v \in V} N^{in}(v)$ respectively.

In contrast to an imperative process language like BPMN, ORG is a declarative description and not necessarily fully specified.

2.2 Process Tree

We want to generate the process model in the form of a process tree (PT). In contrast to a graph-based process model, the process tree has two important characteristics. First, it can be easily transformed into an executable process language, see [17]. Second, a process tree is sound by default [10]. This means the following: First, the process will terminate properly. Second, for each task there is at least one process instance containing it. Each *process tree* $PT = (\mathcal{V}, \mathcal{E})$ is an ordered tree, thus a rooted tree for which an ordering is specified for the children of each vertex. $\mathcal{V}$ consists of leaf nodes $\mathcal{V}_t$ and inner nodes $\mathcal{V}_c$, $\mathcal{V}_t \cup \mathcal{V}_c = \mathcal{V}$, $\mathcal{V}_t \cap \mathcal{V}_c = \emptyset$. Each leaf node corresponds to a task, and each inner node corresponds to a control structure. In this paper we consider three control structures, namely sequence SEQ, parallel AND and exclusive XOR. These control structures correspond to the basic control workflow patterns [2]. This study focuses on the synthesis of process models without cycles. Hence, we do not define a loop operator. It is possible to model the commissioning processes using those control structures. Each control structure can be translated to another block-oriented language, e. g., WS-BPEL, OTX, or to a graph-oriented process language, e. g., Petri nets, BPMN.

2.3 Modular Decomposition

We want to generate a process tree from the declarative specification, i. e., from the ORG. Let $G = (V, E)$ be such a graph. For any $W \subseteq V$ we say that $G_W(V^W, E^W)$ is the sub-graph induced by W, i. e., $V^W = W$ and $E^W = E \cap (W \times W)$. We call W a component iff $\forall v, v' \in W$, $N^{out}(v) \backslash W = N^{out}(v') \backslash W$ and $N^{in}(v) \backslash W = N^{in}(v') \backslash W$. Thus v and v' have identical neighborhoods outside of W. In other words, a component consists of tasks with the same dependencies regarding tasks outside of the component.

***Example* 4.** *The set $\{T, C\}$ is a component of the graph in Figure 3(a). T has incoming edges from E, G, and O and no outgoing edge except the one to C, C shares the same edges, not considering the inner edge between T and C. The set $\{T, G\}$ is not a component because T has an incoming edge from E and G has not.*

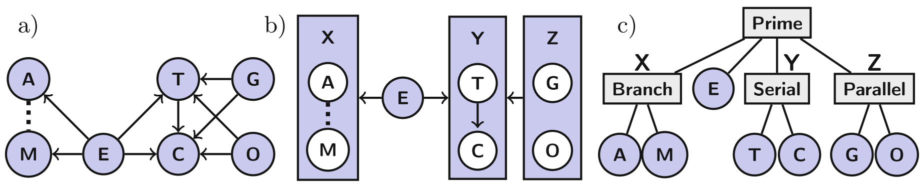


Fig. 3. An Ordering Relation Graph (a), its Modular Decomposition (b) and the Corresponding Modular Decomposition Tree (c)

In our use case, a component often consists of tasks operating on the same electronic control unit of the vehicle. W is a strong component if, for each component $W' \subseteq V$, one of the following holds: $W \cap W' = \emptyset$, $W \subseteq W'$, or $W' \subseteq W$.

***Example* 5.** *Consider a graph G_2 with tree nodes A, B, C and no edges. $W = \{A, B\}$ and $W' = \{B, C\}$ are components. They are not strong because $W \cap W' \neq \emptyset$, $W \not\subseteq W'$, or $W' \not\subseteq W$. The strong components are $\{A, B, C\}$, $\{A\}$, $\{B\}$, $\{C\}$.*

The decomposition of a graph into strong components is called Modular Decomposition, and the resulting hierarchical structure is called Modular Decomposition Tree (MDT). Figure 3(a) shows the simple ordering relation graph of Figure 1, its decomposition in four components Figure 3(b) and the corresponding modular decomposition tree Figure 3(c). [15] shows that a node W in a MDT with children $S_1, S_2, \ldots, S_k$ is of one of the following:

Complete : $\forall I \subset \{1, \ldots, k\}$, with $1 < |I| < k : \bigcup_{i \in I} S_i$ is a component

Prime : $\forall I \subset \{1, \ldots, k\}$, with $1 < |I| < k : \bigcup_{i \in I} S_i$ is not a component

***Example* 6.** *The root node in Figure 3(c) is a prime node. None of the subsets of the children with size 2 or 3, e.g., $\{X, E\}$ or $\{E, Y, Z\}$, do form a component.*

A complete component W with the induced graph $G_W(V^W, E^W)$ either does not contain any edges or is a clique in $E^W_{\#}$ or $E^W_{\rightarrow}$, see the proof of Lemma 1. A complete component can easily be transformed to a process tree deterministically, see [21]. For a prime component our approach will use a heuristic optimization.

Lemma 1. *A strong complete component W is of exactly one of four types:*

trivial : $|V^W| = 1$

serial : *For every $v, v' \in V^W$: $(v, v') \in E^W_{\rightarrow} \lor (v', v) \in E^W_{\rightarrow}$. Recall that the edges in $E^W_{\rightarrow}$ are cycle-free.*

branch : *For every $v, v' \in V^W$: $(v, v') \in E^W_{\#}$*

parallel : *For every $v, v' \in V^W$: $(v, v') \notin E^W$*

Proof. The proofs of all lemmas are in [17]. □

[15] proves that the decomposition of a directed graph (V, E) can be done in $O(|V| + |E|)$, thus in time linear with the size of the graph – We use the MDT to transform the ORG into a process tree.

3 Generating a Process Tree

In this section we explain the conceptual design of our approach. Subsection 3.1 gives an overview, and Subsection 3.2 states how the algorithm handles under-specified regions.

Algorithm 1. Synthesize(ORG G, context c): ProcessTree PT

```
 1: Determine $\mathcal{T}_c$ from $c$
 2: $G \leftarrow$ subgraph $G_W$ of $G$ with the nodes $W = \mathcal{T}_c$
 3: $PT \leftarrow$ Modular Decomposition of $G$
 4: for all prime nodes $\mathcal{P} \in PT$ do
 5:     Process tree PT$_\mathcal{P} \leftarrow$ synPrime($\mathcal{P}$)
 6:     Replace $\mathcal{P}$ with $PT_\mathcal{P}$
 7: end for
 8: for all leaf nodes $l \in PT$ do
 9:     if $l$ is a partition leaf node then
10:         $G_l \leftarrow$ ORG of $l$
11:         Process tree PT$_l \leftarrow$ synthesize($G_l, c$)
12:         Replace $l$ with $PT_l$
13:     end if
14: end for
15: return $PT$
```

3.1 Overview of the Automatic Generation

Our goal is to automatically generate a process model from a declarative description. Algorithm 1 synthesizes a process tree from an ORG and a context c. The context c determines the required tasks $\mathcal{T}_c$ (Line 1). We then reduce the ORG G to the subgraph G_W with the nodes $W = \mathcal{T}_c$ (Line 2). The algorithm then computes a modular decomposition of the ORG (Line 3 in Algorithm 1). The resulting modular decomposition tree (MDT) may contain both complete and prime components. For complete components, a transformation to process fragments exists, cf. [21]. For a prime component in turn, several fragments are possible, see Figure 4. In other words, each prime component stands for an under-specified region. For each prime component $\mathcal{P}$, we use a probabilistic optimization to find a solution (Line 5). We replace $\mathcal{P}$ with the solution found (Line 6). *synPrime()* splits the ORG of the prime components into partitions. It generates a graph with one node for each of these partitions. The algorithm recursively calls itself, in order to replace each node with a subtree. Finally, our approach transforms the PT into a process language, e. g., BPMN, WS-BPEL.

3.2 Under-Specified Regions

Each prime component $\mathcal{P}$ induces a graph $G_\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P})$. $V_\mathcal{P}$ denotes the set of strong components that belong to $\mathcal{P}$. Figure 3 shows that the graph $G_\mathcal{P}$ for the

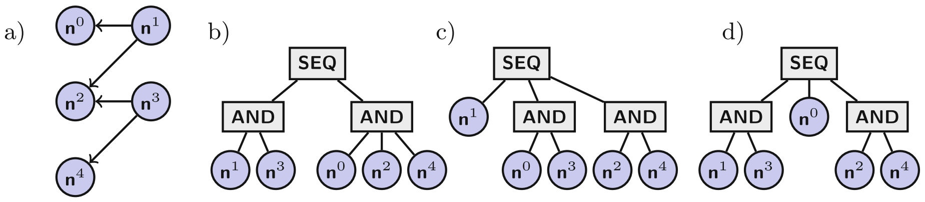


Fig. 4. The Neighborhood Graph to Directly Generate a Process Tree (a), three Possible Process Trees (b)(c)(d) for the Graph (a)

prime component $\mathcal{P}$ consists of $V_{\mathcal{P}} = \{\mathsf{X}, \mathsf{R}, \mathsf{Y}, \mathsf{Z}\}$ with $E_{\mathcal{P}} = \{(\mathsf{R} \rightarrow \mathsf{X}), (\mathsf{R} \rightarrow \mathsf{Y}), (\mathsf{Z} \rightarrow \mathsf{Y})\}$. $\mathcal{P}$ is not fully specified and thus there does not exist a unique corresponding process tree. Due to the large number of possible process models for a prime graph $G_{\mathcal{P}}$ it is not feasible to construct every possible one.

The modular decomposition detects the fully specified and the under-specified regions of the process. Our overall idea is to reduce the size of the graph induced by a prime component iteratively until the number of remaining solutions is low (< 100) so that we can solve the problem. See Figure 4. Our intuition for the reduction is to select a pivot node v and detect which nodes (V_1) must occur before v, and which nodes (V_2) can be scheduled in parallel to v. V_1 as well as V_2 imply two smaller ordering graphs. We repeat this with several different pivot nodes. Our approach randomly selects a node $v \in V_{\mathcal{P}}$ with $N^{out}(v) = \emptyset$ as pivot node. Lemma 3 will show why we need this characteristic. The ORG $G_{\mathcal{P}}$ is cycle-free, and thus a node v with $N^{out}(v) = \emptyset$ always exists.

Definition 3. *The zero neighborhood of a pivot node v is $N^{(0)}(v) := \{v\}$, $N^{(1)}(v) := N^{in}(v)$. For $i \in \mathbb{N}$, $i > 1$ we define the i-neighborhood as:*

$$N^{(i)}(v) := \begin{cases} \left(\bigcup_{v' \in N^{(i-1)}(v)} N^{out}(v')\right) \setminus \; N^{(i-2)}(v) & \text{if } i \in \{2, 4, 6, \dots\} \\ \left(\bigcup_{v' \in N^{(i-1)}(v)} N^{in}\ (v')\right) \setminus \; N^{(i-2)}(v) & \text{if } i \in \{3, 5, 7, \dots\} \end{cases}$$

We use the neighborhood information to partition the graph. Each partition $n^{(i)}$ is a subgraph of the ORG $G_{\mathcal{P}}$ with the nodes $N^{(i)}(v)$. In other words, the partitioning implies a graph G_v where each $n^{(i)}$ is a node. We refer to this graph as the neighborhood graph. Formally, given a pivot node v, the neighborhood graph $G_v = (V_v, E_v)$ is as follows

$$\begin{aligned} V_v &= \{n^i \mid N^{(i)}(v) \neq \emptyset\} \\ E_v &= \{(n^i, n^{i+1}) \mid i \in \{1, 3, \dots\} \wedge n^i, n^{i+1} \in V_v\} \cup \\ &\quad\ \{(n^{i+1}, n^i) \mid i \in \{0, 2, \dots\} \wedge n^i, n^{i+1} \in V_v\} \end{aligned}$$

The graph contains each non-empty neighborhood as a node.

Example 7. *For the graph in Figure 3(b) and the pivot Y the neighborhoods are: $N^{(0)}(\mathsf{Y}) = \{\mathsf{Y}\}$, $N^{(1)}(\mathsf{Y}) = \{\mathsf{R}, \mathsf{Z}\}$, $N^{(2)}(\mathsf{Y}) = \{\mathsf{X}\}$, and for $i > 2$ $N^{(i)}(\mathsf{Y}) = \emptyset$.*

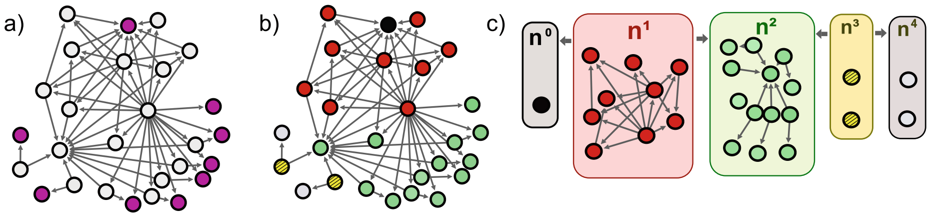


Fig. 5. A Prime Component (a), its Partitioning (b), and the Neighborhood Graph (c)

The neighborhood graph $G_\mathsf{Y}(V_\mathsf{Y}, E_\mathsf{Y})$ for the pivot Y *is:*

$$G_\mathsf{Y} = (\ \{n^0,\ n^1,\ n^2\}\ ,\ \{\ (n^1,\ n^0)\ ,\ (n^1,\ n^2)\ \})$$

***Example* 8.** *Figure 5(a) shows a more complex graph which is a prime component, i. e., there is no unique corresponding tree. The possible pivot nodes are in violet. The pivot node at the top of Figure 5(a) leads to the partitioning in Figure 5(b). Figure 5(c) shows the respective neighborhood graph.*

Lemma 2. *The partitioning into the neighborhood graph for a pivot v preserves all order dependencies. In other words, for each edge $(v_1, v_2) \in E_\mathcal{P}$, one of the following holds:*

(a) $\exists i \in \mathbb{N}_0 : v_1, v_2 \in N^{(i)}(v)$
(b) $v_1 \in N^{(i)}(v), v_2 \in N^{(j)}(v), i \neq j \Rightarrow (n^i, n^j) \in E_v$

Lemma 2 states that our approach does not loose any dependencies. A symmetric solution would be to select pivots with $N^{in}(v) = \emptyset$ and change the definition of the neighborhood accordingly. However, a pivot v with $N^{out}(v) \neq \emptyset \ \wedge \ N^{in}(v) \neq \emptyset$ would loose a dependency, see Lemma 3.

Lemma 3. *The neighborhood graph G_v for a pivot node v with $N^{out}(v) \neq \emptyset \wedge N^{in}(v) \neq \emptyset$ does not preserve the order dependencies.*

Algorithm 2. synPrime (Neighborhood Graph $G(V, E)$) : ProcessTree PT

```
1: Pivot v ← randomly select a node $v \in G$ with $N^{out}(v) = \emptyset$
2: $G_v(V_v, E_v)$ ← calculate neighborhood of $v$
3: if $N^{(\lambda)} = \emptyset$ then
4:     return (select tree pattern randomly)
5: else
6:     return $synPrime(G_v)$
7: end if
```

Algorithm 2 generates a process tree for an under-specified region, i. e., a prime component. First, the algorithm randomly selects a pivot node v (Line 1) and calculates its neighborhood graph G_v (Line 2). The parameter $\lambda \in \mathbb{N}^+$ defines when the neighborhood graph is small enough to generate a process tree. If the neighborhood graph is too large, the algorithm calls *synPrime* again, and everything is repeated until the graph is processable. Figure 6 shows the reduction of a neighborhood graph. If our approach selects n^2 as the pivot element, it then builds the smaller graph on the right hand side.

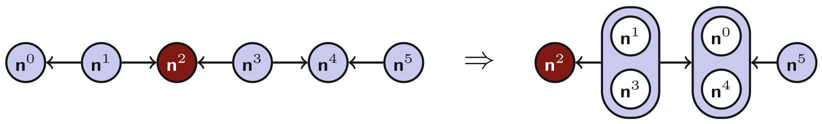


Fig. 6. Reduction of a Neighborhood Graph with the Pivot n^2

If the neighborhood graph is small enough ($N^{(\lambda)} = \emptyset$), Algorithm 2 randomly selects a tree pattern for it (Step 4). A tree pattern is a process tree for the neighborhood graph. The neighborhood graph in Figure 4(a) contains 5 nodes and 4 edges. For a graph with five nodes thousands of process trees are possible. For the graph in Figure 4(a) 53 trees are possible, given the constraints. For most of these 53 process trees, there is another tree with a lower overall processing time, for any processing times of the tasks. If we exclude these dominated trees, three trees remain. Figures 4(b) and (c) show two of them, randomly selected. The tree patterns define which additional dependencies have to be added to generate a block based process model for the specification. [17] shows and explains all tree patterns for $\lambda \in [1, 5]$. Figure 4(d) shows a process tree fulfilling the constraints in Figure 4(a), but the processing time of the tree in Figure 4(b) always is shorter.

For each ORG we have started out with, we calculate κ different process trees. The resulting trees differ depending on the probabilistic choices in Algorithm 2 (Line 1) and (Line 4). We select the best process tree found according to quality criteria, e. g., the processing time. We calculate a quality value of each tree as follows. The average processing time for each node in a process tree $PT(\mathcal{V}, \mathcal{E})$ is calculated recursively with function *fit*: $\mathcal{V} \to \mathbb{R}$.

$$
\mathit{fit}(n) := \begin{cases} \mathit{runtime}(n) & \text{if } \mathit{type}(n) = \mathit{task} \\ \max_{c \in \mathit{child}_n} \mathit{fit}(c) & \text{if } \mathit{type}(n) = \text{AND} \\ \sum_{c \in \mathit{child}_n} \mathit{fit}(c) & \text{if } \mathit{type}(n) = \text{SEQ} \\ \max_{c \in \mathit{child}_n} \mathit{fit}(c) & \text{if } \mathit{type}(n) = \text{XOR} \end{cases}
$$

type: $\mathcal{V} \to \{\mathit{task}, \text{AND}, \text{SEQ}, \text{XOR}\}$ is a function to determine the type of the tree node. $\mathit{child}_n := \{c \mid (n, c) \in \mathcal{E}\}$ is the set of nodes in the process tree with parent node n. The estimation for the XOR-Split is a worst case analysis, i.e., the processing time is smaller than the estimated one. If the probabilities of the splits are known a priori a more precise average case assumption is possible,

Table 1. Computation Time (CT) and Processing Time (PT) of our Approach

	Process A		Process B		Process C	
No. of Tasks	171		185		116	
Ref. process time	171 780ms		169 606ms		148 014ms	
	CT in ms	PT in ms	CT in ms	PT in ms	CT in ms	PT in ms
10 Iterations	35	188,420	34	227,260	**32**	**132,998**
50 Iterations	**69**	**127,687**	**71**	**131,121**	66	103,234
100 Iterations	113	127,687	113	131,121	104	103,234
1 000 Iterations	823	127 687	964	116,155	788	97,264
10,000 Iterations	8,207	112,918	8,298	113,874	7,817	71,513
100,000 Iterations	78,409	112,624	86,594	106,216	77,335	65,892
PT reduction	34.437 %		37.375 %		50.456 %	

see [24]. The fitness of a process tree *fit*(PT) is the fitness of its root node. The algorithm returns the process tree with the highest fitness value. The resulting process tree can easily be transformed to the notation required.

We have implemented the algorithms in C#. The program receives the ORG as input, see [17] on how to generate an ORG from a declarative specification. The output of the program is a process tree that is then transformed to the commissioning process notation OTX by a proprietary XSLT script written by us. The implementation can handle specifications with several hundreds of tasks and thousands of dependencies in a few minutes, see Section 4.

4 Evaluation

Our evaluation uses 21 process models from a car manufacturer that specify the testing and commissioning of middle-class vehicles. Each process model reflects several context characteristics which are attached for the generation. The context characteristics consist of properties of the vehicle project, of the factory and of the components to put in commission. Professional process developers have designed the process models. The tasks to be executed depend on the components built into the vehicle to be tested. In cooperation with those domain experts we have built the specification for the 21 process models, i. e., the ordering relationship graphs, automatically using a knowledge base. See [19]. The process models contain up to 185 tasks and over 3000 dependencies, including transitive ones. The parameter λ defines the maximum size of the process trees. The possible number of trees grows exponentially with the maximum size. Therefore, the correct and optimal tree patterns are harder to find for larger values of λ. Otherwise, a higher value could allow to find a process model with a better processing time. We choose $\lambda = 5$ for our evaluation.

Table 1 shows the results for commissioning process models A, B, and C. We have chosen A, B, and C because they are representative for the whole set, ranging from a relatively small one (C) to one of the largest (B). For a summary

Table 2. The Minimum, Maximum, and the Quartile for the Evaluation of 21 Commissioning Process Models

	Minimum $Q_{0.00}$	$Q_{0.25}$	Median $Q_{0.5}$	$Q_{0.75}$	Maximum $Q_{1.00}$
Nr. of Tasks	98	123	133	146	185
Ref. Process Time	144.232s	151.623s	157.513s	166.138s	178.606s
best found pt	64.643s	72.637s	84.529s	93.487s	108.496s
iterations (it)	5 090	15 523	37 733	76 035	94 271
calculation time (ct)	4.641s	12.356s	30.284s	63.392s	77.480s
pt reduction	33.39%	40.62%	47.54%	53.62%	58.03%

of all models see Table 2. The second row in Table 1 shows the processing time measured for the process model created by hand. Table 1 then lists the expected processing time of the process (pt) and the time our approach needs to generate the respective model (computation time ct) for 10 to 100,000 iterations. In all cases, the algorithm has been able to generate a process model in less than 100 ms that outperforms the reference process model. After 100,000 iterations (in less than 1.5 minutes) it could find process models with processing times 34%, 37%, and 50% lower than their manually generated counterparts.

For all 21 process models, Table 2 shows the minimum, maximum, and the quartile for 7 values of the evaluation. The process models contain between 98 and 185 tasks, and need up to 178s to perform. Our approach requires $\approx 30s$ and $\approx 37\,000$ iterations on average to generate the best result found. For all instances our approach has identified a solution that is better than the manually generated one in less than 100 ms. Our approach needs less than 3 iterations to do so in most cases. On average, it nearly halves the processing time of the commissioning process models (47.47%) compared to the reference points.

5 Related Work

[25] synthesizes a process model directly from its specification. The specifications are in propols [25], a temporal constraint specification language. The specifications are transformed into finite state machines and then integrated into one machine. Next, each accepting path is generated from the state machine. An algorithm similar to the α-algorithm [3] is applied to synthesize a process model from its set of paths. [25] can only be applied if the specification, i. e., the number of state machines, is small ($\approx$ 6). To this end, [25] divides the specification into small groups, synthesizes a process fragment for each group and manually combines the fragments. For our use case, this approach would require over a hundred state machines for each commissioning process model, and the manual combination would not be feasible. [6] has specifications with ltl as starting point. It generates a pseudo model from the specification. This model lists all paths that fulfill the ltl formula. [6] generates an ordering relation graph from

the set of paths and uses it to synthesize a process tree. For our use case the generation of all paths would not be feasible. This is because the number of paths grows exponentially with the size of the specification. Even for the smallest process model we have evaluated calculating all paths has not been possible.

Process discovery means finding a process model that can reproduce the behavior given in a log [1]. [12] rediscovers a process model in the process-tree notation. It generates a graph (directly-follows graph) from the log and tries to find different kinds of cuts in the graph. Each kind of cut refers to a control structure in the process tree (SEQ, AND, XOR, LOOP). The cuts partition the graph and allow to hierarchically find a process tree for the log. In contrast to an ORG, a directly-follows graph is not transitive, and if two nodes are in parallel they share a two-way edge (no edge in the ORG). It is not possible to find a cut for a prime component, thus the approach of [12] does not help in case the specification is under-specified. Put differently, the problem statement in [12] is different from ours; the neighborhood graph of the complete log of a process tree never contains a prime component. For an incomplete log, a prime component can occur. [13] proposes to use probabilistic activity relations in the case of an incomplete log. The cut with the highest probability is chosen. This means that their algorithm generalizes from the incomplete log and assumes relationships that are not present. An ORG is an upper bound of the possible behavior. Assuming an additional relation would result in a violation of a constraint.

An approach different from generating the process model from scratch is to extract information from process models already specified and to create a similar process. [7] uses a CBR-based method to this end. The search is based on keywords that are annotations of the workflows. [9] guides the process designer with suggestions on how to complete data-oriented visualization models. The suggestions are generated from paths of existing visualization process models stored in a repository. [9] does not allow building a process model with an AND-Split and therefore is not sufficient in our case. [11] predicts which activity pattern (generic process fragment) will follow the partly modeled process. The paths of existing process models are extracted and analyzed with association rule mining. [9][11] extend an existing process model, while our approach generates one from a declarative specification. [7] requires annotations of the existing process models. None of the approaches mentioned optimize the runtime or consider constraints.

[21] transforms an unstructured model without cycles into a behaviorally equivalent structured process model. 'structured' means that for each Split-Gateway there is a corresponding Join-Gateway. Structured processes allow an effective verification [18] and are easy to understand [22]. [21] determines relationships between the tasks of a process model and generates an ORG using these relationships. Next, [21] decomposes the ORG into a Modular Decomposition Tree. In contrast to our approach, [21] generates the ORG from the behavior of an existing process model and not from a set of compliance rules. The behavior is definite, the result therefore is a unique process model. In our approach in turn, the behavior is under-specified, and several process models are possible.

AI planning is the task of defining a set of actions that achieve a specified aim [8]. In a nutshell, it is the search for an applicable plan in the solution space.

[23] uses a genetic algorithm to find a manufacturing plan. Some approaches that synthesize business processes are discussed next: [14] uses an AI planning approach to synthesize service compositions. Without calling it AI planning, [4] uses a similar approach for configuration-based workflow composition. [5] introduces a planning algorithm to compose data workflows. None of these studies focuses on optimizing the runtime of the process or considers requirements similar to ours. These approaches are not applicable to our problem statement.

In contrast to imperative process models, declarative workflows allow for any behavior fulfilling the declarative specification [16]. Thus, declarative workflows provide maximum flexibility not limited by a process model. In comparison, [25], [6] and our approach generate an imperative process model from the declarative specification. The enactment of declarative workflows is not trivial [20], and tool support by major vendors is missing. To our knowledge, there is no tool that executes declarative process models comparable to the commissioning of vehicles.

6 Conclusions

We have proposed a novel approach to generate a process model for a specific context automatically, given a set of constraints. We study the restricted case that there are not any repetitions of a task, as is the case in commissioning and elsewhere, e.g., manufacturing. We use a probabilistic search to find a good process model according to quality criteria that fulfills the constraints. Our approach can handle complex real-world specifications consisting of several hundred constraints and more than one hundred tasks. The process models generated with our scheme are superior (nearly twice as fast) to ones designed by professional process designers.

In future work we want to omit the cycle free limitation of process models. One approach could be to detect SESE (**S**ingle **E**ntry **S**ingle **E**xit) loops in the graph, similarly to [21] Chapter 6.4. One could also extend the approach to resource dependencies limiting the possible number of parallel executions of certain tasks. One could detect such situations analyzing the graph structure and then add additional dependencies for the generation.

Acknowledgments. This research is supported by AUDI AG.

References

1. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer Publishing Company, Incorporated, 1st edn. (2011)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases **14**(1), 5–51 (2003)
3. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16**(9), 1128–1142 (2004)
4. Albert, P., Henocque, L., Kleiner, M.: Configuration based workflow composition. In: IEEE International Conference on Web Services, vol. 1, pp. 285–292, July 2005

5. Ambite, J.L., Kapoor, D.: Automatically composing data workflows with relational descriptions and shim services. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 15–29. Springer, Heidelberg (2007)
6. Awad, A., Goré, R., Thomson, J., Weidlich, M.: An iterative approach for business process template synthesis from compliance rules. In: Mouratidis, H., Rolland, C. (eds.) CAiSE 2011. LNCS, vol. 6741, pp. 406–421. Springer, Heidelberg (2011)
7. Chinthaka, E., Ekanayake, J., Leake, D., Plale, B.: CBR based workflow composition assistant. In: IEEE World Conference on Services, pp. 352–355, July 2009
8. Hendler, J., Tate, A., Drummond, M.: AI Planning: Systems and Techniques. Tech. rep., University of Maryland at College Park, College Park, MD, USA (1990)
9. Koop, D., Scheidegger, C., Callahan, S., Freire, J., Silva, C.: VisComplete: Automating Suggestions for Visualization Pipelines. IEEE Transactions on Visualization and Computer Graphics **14**(6), 1691–1698 (2008)
10. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems Architecture **4**(1), 3–13 (2009)
11. Lau, J.M., Iochpe, C., Thom, L., Reichert, M.: Discovery and analysis of activity pattern cooccurrences in business process models. In: Int'l Conf. on Enterprise Information Systems, Milan, Italy, pp. 83–88, May 2009
12. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013)
13. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from incomplete event logs. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014. LNCS, vol. 8489, pp. 91–110. Springer, Heidelberg (2014)
14. Matskin, M., Rao, J.: Value-added web services composition using automatic program synthesis. In: Bussler, C.J., McIlraith, S.A., Orlowska, M.E., Pernici, B., Yang, J. (eds.) CAiSE 2002 and WES 2002. LNCS, vol. 2512, pp. 213–224. Springer, Heidelberg (2002)
15. McConnell, R.M., de Montgolfier, F.: Linear-time modular decomposition of directed graphs. Discrete Applied Mathematics **145**(2), 198–209 (2005)
16. Montali, M., Pešić, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies. ACM Trans. Web **4**(1), 3:1–3:62 (2010)
17. Mrasek, R., Mülle, J., Böhm, K.: Automatic Generation of Optimized Process Models from Declarative Specifications. Technical Report 2014–15, KIT, Karlsruhe, November 2014. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000044586
18. Mrasek, R., Mülle, J., Böhm, K.: A new verification technique for large processes based on identification of relevant tasks. Information Systems (2014)
19. Mrasek, R., Mülle, J., Böhm, K., Becker, M., Allmann, C.: User-friendly property specification and process verification – a case study with vehicle-commissioning processes. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 301–316. Springer, Heidelberg (2014)
20. Pešić, M., Bošnački, D., van der Aalst, W.M.P.: Enacting declarative languages using LTL: avoiding errors and improving performance. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 146–161. Springer, Heidelberg (2010)

21. Polyvyanyy, A.: Structuring Process Models. University of Potsdam, Potsdam (2012)
22. Reijers, H., Mendling, J.: A Study Into the Factors That Influence the Understandability of Business Process Models. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans **41**(3), 449–462 (2011)
23. Váncza, J., Márkus, A.: Genetic algorithms in process planning. Computers in Industry **17**(2–3), 181–194 (1991)
24. Yang, Y., Dumas, M., García-Bañuelos, L., Polyvyanyy, A., Zhang, L.: Generalized aggregate Quality of Service computation for composite services. Journal of Systems and Software **85**(8), 1818–1830 (2012)
25. Yu, J., Han, Y.B., Han, J., Jin, Y., Falcarin, P., Morisio, M.: Synthesizing Service Composition Models on the Basis of Temporal Business Rules. Journal of Computer Science and Technology **23**(6), 885–894 (2008)