Process Synthesis with Sequential and Parallel Constraints

Richard Mrasek, Jutta Mülle, and Klemens Böhm

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

Abstract. Synthesis is the generation of a process model that fulfills a set of declarative constraints, a. k. a. properties. In this article, we study synthesis in the presence of both so-called sequential and parallel constraints. Sequential constraints state that certain tasks must occur in a specific ordering. Parallel constraints specify the maximal degree of parallelization at a certain position in a process model. Combining both sequential and parallel constraints in one approach is difficult, because their interference is complex and hard to foresee. Besides this, with large specifications, solutions which do not scale are not viable either. Our synthesis approach consists of two steps. First, we generate a model fulfilling only the sequential constraints. We then apply a novel algorithm that deparallelizes the process to fulfill the parallel constraints as well as any additional optimization criteria. We evaluate our approach using the real-world use case of commissioning in vehicle manufacturing. In particular, we compare our synthesized models to ones domain experts have generated by hand. It turns out that our synthesized models are significantly better than these reference points.

1 Introduction

Synthesis is the generation of a process model that fulfills a set of declarative constraints, a. k. a. properties. Synthesis has several important application scenarios, described in various publications:

- to generate a process skeleton as a basis for process models or as a communication point between business and compliance experts [1],
- to build web service composition models [2],
- to automatically generate production processes [3],
- to assure that the process model complies with life cycles of its artifacts [4].

Synthesis is different from verification, but is related [5][6]. Verification is checking whether a given process model has certain properties. Synthesis in turn generates a process model from properties. Both require a formal model of the properties as a starting point. Synthesis from compliance rules is also known as *Compliance by Design*[4], in contrast to *Compliance by Validation* [7].

As [1][2][3][4] show for synthesis and [8][9][10] for verification, different types of constraints for process models exist. In this article, we study synthesis in the presence of both so-called sequential and parallel constraints. Sequential constraints state that certain tasks in a process model must occur in a specific ordering. For instance, Task A requires another Task B as its precondition. Parallel constraints specify the maximal degree of parallelization of tasks at a certain position of a process model. They allow modeling any resource limitations. For instance, a Resource R has a capacity of 2, i. e., only two tasks can access it at a time. To avoid blocking, one wants to limit the degree of parallelization of the tasks using the resource to 2. Another requirement is that any synthesis approach must be scalable and cope with specifications and process models of realistic size.

The related work on process synthesis does not support these requirements in combination. [1] and [3] only support sequential constraints. [2] and [4] can handle parallel constraints in principle, but only with an exponential bloating of their finite state automata. [9], [2] and [4] explore the full state space of possible solutions and thus can not cope with large models either.

Challenges Synthesis from both sequential and parallel constraints is difficult, because their interference is complex and hard to foresee. Besides this, with large specifications, solutions which do not scale are not viable either. For instance, the scenario we use in our evaluation, a real one, contains 496 tasks, and there are several hundred sequential and parallel constraints. There often is a huge number of models that fulfill the specification. To illustrate, the sequential arrangement of n nodes, in the absence of any constraint, gives way to $n!$ different process models. Next, orthogonally to the dependencies that a process model must fulfill, several optimization criteria can exist. The total processing time of the process model is just one of them.

Contributions In this paper we propose an approach for the synthesis of process models from both sequential and parallel constraints. To this end, we first collect and classify the requirements that the process model should fulfill, see Section 2. Our collection contains new and interesting requirements not featured by other synthesis approaches. This includes new criteria for correctness (*parallel constraint*, *total processing time*) and secondary optimization constraints. For the synthesis we propose a two-step approach. First, we generate a model fulfilling the sequential constraints, see Section 3. In Section 4, we then apply a novel algorithm that deparallelizes the process to fulfill the parallel constraints as well as the optimization criteria. We evaluate our approach using the real-world use case of commissioning in the vehicle industry, see Section 5. In particular, we compare our synthesized models to ones domain experts have generated by hand. Our synthesized models are significantly better than these reference points. Then, we discuss related work of our approach in Section 6. Finally, we conclude that our approach can indeed deal with large specifications occurring in real scenarios.

2 Process-Model Requirements

In this section we discuss the requirements that a synthesized process model should meet. First, the process model must be correct. Correctness means fulfilling all given properties. Second, the process model should be of high quality, i. e., good with respect to the optimization criteria given, e. g., total runtime – Subsection 2.1 describes different correctness criteria, using our use case as an illustration. Subsection 2.2 introduces optimization criteria, allowing to differentiate between models that are correct.

2.1 Correctness

As mentioned earlier, the form of the constraints in our context can vary. [1] and [2] use different property patterns that can be expressed as temporal logic formulae. [4] uses a state space model of data artifacts, and [3] uses a dependency graph to express sequential constraints.

We want to synthesize process models in the form of process trees. The rationale is that the block structure ensures important syntactic properties, e. g., *soundness*, and the transformation to any process language is easily doable. Additionally, the block structure allows the inclusion of both sequential and parallel constraints. Related work [11] has mainly focused on the use of sequential constraints, subsequently abbreviated with $\mathcal{S}$, see also [9][10][8]. But parallel constraints $\mathcal{P}$ are equally important. This is because every resource has a limited capacity, and effective resource usage is key to business-process efficiency. Finally, constraints on process execution as a whole, like bounds on the total processing time, are worthwhile as well. We have learned from domain experts in industry that these constraints influence the design of process models very much.

Sequential Constraints $\mathcal{S}$. *Sequential constraints* limit the sequential arrangement of tasks that are possible. This includes that a Task A needs another Task B as pre- or post-condition. We consider two patterns of sequential constraints, namely *response* and *precedence*. They represent the fundamentally different types of sequential constraints between tasks, in particular also in our application domain, the commissioning of vehicles. 'A response-to B' means that after the execution of A, B needs to be executed before the process completes. B does not need to be the next task. 'B precedence A' means that, B needs to be executed before Task A. In both patterns, A is the *antecedent*, i. e., the task B is referring to, because it triggers the pattern, and B is the *consequence*.

Example 1. *Our evaluation scenario is the commissioning of vehicles. Commissioning is the final step in the production of a vehicle and consists of testing whether the components function properly and of configuring the electronic components. To this end, a Task A communicates with a specific component built in the vehicle. Before this communication is possible, another Task B needs to establish the connection to the component. The constraint A* precedence *B states this.*

Parallel Constraints $\mathcal{P}$. Parallel constraints limit the parallelization of tasks. This is often necessary because tasks use resources with a limited capacity. In order to avoid blocking of the process execution, the degree of parallelization in the process model should not exceed the capacity of the resource.

Example 2. *With vehicle commissioning, one class of tasks is to configure the software installed on the electronic control units (*ECU*s). This causes a huge volume of data transfer over a bus protocol with a low data rate. These configuration tasks are the most time consuming ones of a commissioning process. The concrete number of parallel connections available for data transfer is four.*

Bounds on Total Processing Time. Additional constraints are possible that relate to the execution as a whole. One such constraint limits the total processing time, i. e., assures that all process instances will complete within this time span.

Example 3. *In our scenario, the vehicle production uses an assembly line system. Each production step is planned to use a fixed number of cycles. After the cycle, the vehicle goes to the next production station. This allows a continuous flow of production and a high productivity. If the completion of the process model needs more time than scheduled, this will negatively influence the flow of production and cause high costs. To this end, the processing time of a process model must not exceed the time limit of the station.*

2.2 Optimization

Even when all correctness criteria are met, many process models are still possible. To this end, it is necessary to select a good model out of the correct ones. The possible criteria are manifold and depend on the scenario. A common criterion is to minimize the average processing time of the process instances.

Example 4. *In our evaluation scenario, two criteria are relevant. Some tasks are executed manually by a worker, and some are executed automatically, i. e., by the control units in the vehicle. The first criterion is to reduce the waiting time of the workers, i. e, the worker should not need to wait until an automated task completes. Following the lean production paradigm, reduction of the human waiting time is more important than the reduction of the processing time [12, p. 54] (as long as the total time limit is met).*

3 Generating a Process Model from Sequential Constraints

The first step of our approach is to find a process model that fulfills the sequential constraints, e. g., the response and precedence pattern. The input parameter of the algorithm is a list of necessary tasks $\mathcal{T}$ and the sequential constraints $\mathcal{S}$. The algorithm consists of the following steps. Subsections 3.1–3.3 describe these steps.

1. Generate a complete dependency graph
2. Decompose the graph into its modules
3. Transform the decomposition into a process model.

3.1 Generating Complete Dependency Graph

In this paper we use the *declare* notation [13] for the graphical representation of sequential constraints, i. e., nodes represent tasks, edges sequential constraints, and a black dot marks the antecedent of the constraint, see Figure 1.

Fig. 1. The Graphical Representation of the Response- and the Precedence-Pattern

Definition 1. [Dependency Graph DP] *A dependency graph is a directed acyclic graph $DP(V, E)$, as follows: The nodes V represent the tasks in the process model. An edge $(v_1, v_2) \in E$ represents a sequential constraint between Tasks v_1 and v_2.*

The graph needs to be acyclic. Otherwise, it would not be possible to find a correct process model. The dependency graph should not be confused with the control flow graph modeled by an imperative process model, e. g., BPMN. The dependency graph constrains the process execution but does not explicitly state how to execute the model.

Algorithm 1, which we have published in a technical report earlier [14], shows our approach to generate the dependency graph. We assume, that their exist no contradictionary constraints. $\mathcal{S}$ may refer to tasks that are not in $\mathcal{T}$, so we extend $\mathcal{T}$ with these tasks in a first step. For instance, think of a process model including Task X. X requires another Task Y as its precondition, thus we must include it in the process model as well. Given $\mathcal{T}$ and $\mathcal{S}$, we check if a task $t \in \mathcal{T}$ is the antecedent of a sequential constraint with a task t_2 not in $\mathcal{T}$ as the consequence. If so, we add t_2 to $\mathcal{T}$. Our algorithm repeats these steps until no further change is possible (Line 1-3). For each task we generate a node in the dependency graph (Line 4-6). If a sequential constraint between two tasks $t_1, t_2 \in \mathcal{T}$ exists, we add an edge (t_1, t_2) to the dependency graph (Line 7-11). At last, we generate the transitive hull of the dependency graph (Line 12).

Example 5. *Figure 2 illustrates different states of the algorithm. The tasks in $\mathcal{T}$ in the respective state are highlighted in red. The algorithm starts with the initial set of tasks $\mathcal{T} = \{B, E\}$ and the sequential constraints $\mathcal{S}$ containing six constraints according to the edges; see Figure 2(a). Next, we extend $\mathcal{T}$ to all tasks that are the consequence to an antecedent task in $\mathcal{T}$. For instance, C is consequence in the constraint 'E precedence C'. See Figure 2(b). We repeat this step until no change happens, see Figure 2(c). F is not in $\mathcal{T}$ because F is antecedent in the pattern. Next, we generate the dependency graph using $\mathcal{T}$ and $\mathcal{S}$. Figure 2(d)*

Algorithm 1 GenerateDPGraph (task set $\mathcal{T}$, Sequential Constraints $\mathcal{S}$) : DP

while $\exists t_1 \in \mathcal{T},\ t_2 \notin \mathcal{T}$ with $(t_1, t_2) \vee (t_2, t_1) \in \mathcal{S}$ and t_1 is the antecedent part **do**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t_2\}$
end while
for all $t \in \mathcal{T}$ **do**
    Add a state t to the dependency graph DP
end for
for all $(t_1, t_2) \in \mathcal{S}$ **do**
    if $t_1, t_2 \in \mathcal{T}$ **then**
        Add an edge (t_1, t_2) to the dependency graph DP
    end if
end for
$DP \leftarrow$ Build the transitive hull of DP
return The dependency graph DP

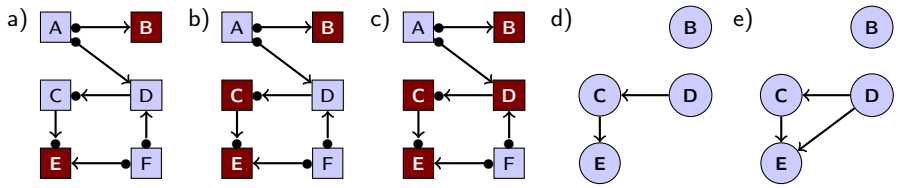


Fig. 2. Generation of a Complete Dependency Graph

shows the resulting graph. Lastly, we build the transitive hull of the dependency graph, shown in Figure 2(e).

3.2 Modular Decomposition

In general the dependency graph is complex, containing hundreds of nodes and edges. To this end, we decompose the graph into manageable modules to facilitate the synthesis.

Definition 2. [Set of Outgoing Vertices $N^+(v)$ and Incoming Vertices $N^-(v)$] *The set of outgoing vertices $N^+(v)$ of a vertex v in a directed graph $G(V, E)$ is $N^+(v) = \{v' | (v, v') \in E\}$. $N^-(v) = \{v' | (v', v) \in E\}$ is the set of incoming vertices.*

Definition 3. [Module] *A module W for a directed graph $G(V, E)$ is a subset of the vertices $W \subseteq V$ where all vertices $v \in W$ have the same incoming $N^-(v)$ and outgoing vertices $N^+(v)$ not in W.*

Definition 4. [Strong Module] *A module W is a strong module if, for each module $W' \subseteq V$, one of the following holds: $W \setminus W' = \emptyset$, $W \subseteq W'$, or $W' \subseteq W$.*

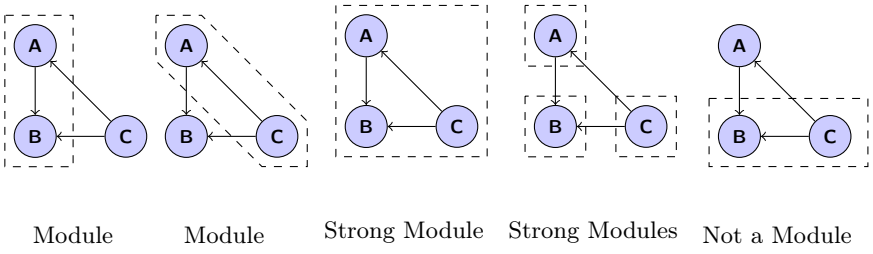


Fig. 3. A Dependency Graph with Some Modules

Example 6. *The first four images in Figure 3 show all modules for a simple dependency graph. The set of all modules is* $\mathcal{M} = \{\{A, B\}, \{A, C\}, \{A, B, C\}, \{A\}, \{B\}, \{C\}\}$. $\{A, B\}$ *and* $\{A, C\}$ *are not strong modules. This is because* $\{A, B\} \setminus \{A, C\} = \{A\} \neq \emptyset$ *and* $\{A, B\} \not\subseteq \{A, C\}$ *and* $\{A, C\} \not\subseteq \{A, B\}$. $\{B, C\}$ *is not a module because* $N^-(B) \setminus \{B, C\} = \{A\} \neq \emptyset = N^-(C) \setminus \{B, C\}$.

The strong modules do not intersect with each other, i. e., each module is either part of another module, or no shared vertex exists. This allows decomposing the graph into a hierarchy of strong modules called the modular decomposition tree. The generation of the modular decomposition tree can be done in linear time [15].

Definition 5. [Module Classification] *A strong module* W *is either prime, i. e., no real subset* $W' \subseteq W$ *with* $|W| > 1$ *is a module, or complete, i. e., all subsets are modules.*

[15] shows that each strong module is either prime or complete. In this paper we assume that the decomposition results in a complete module. In general, prime modules are parts of a process model that is not fully specified. [3] proposes an approach to solve such under-specified parts and to transform them into complete modules.

3.3 Transform into Process Tree

We classify complete modules into three types:

Definition 6. *The following three types of a complete module* W *exist:*

- Trivial*:* $|W| = 1$
- Parallel*:* $\forall v_1, v_2 \in W, v_1 \neq v_2 : (v_1, v_2) \notin E$
- Serial*:* $\forall v_1, v_2 \in W : (v_1, v_2) \in E \vee (v_2, v_1) \in E$

Lemma 1. *A complete module* W *is exactly of one of the types* trivial*,* parallel*, or* serial*.*

Proof. It is obvious that a module W cannot be of more than one of these types. So it remains to show that a complete module is always either *trivial*, *parallel* or *serial*. For $|W| = 1$ the module is *trivial*. For $|W| = 2$, either an edge exists between the two vertices, i. e., the module is *serial*, or not, i. e., the module is *parallel*.

We now show the general case by contradiction. To this end, next to $|W| > 2$, we assume that W is neither *parallel* nor *serial*. This means that $\exists v_1, v_2 \in W : (v_1, v_2) \in E$ and $\exists v_3, v_4 \in W : (v_3, v_4) \notin E \wedge (v_4, v_3) \notin E$. We partition the module into $W' = \{v_1, v_3)$ and $W'' = \{v_2, v_4\}$. W is a *complete* module, implying that W' and W'' are modules. Since v_1 and v_3 are in a module, they have the same outgoing edges outside of W'. Analogously, v_2 and v_4 are in a module, and they share the same incoming edges to nodes outside of W''. An edge $(v_1, v_2) \in E$ exists, therefore an edge $(v_3, v_4) \in E$ exists. This is a contradiction to the assumption that the module is not *serial*. □

Each type corresponds to a control-flow element in the process tree: the *trivial* modules to tasks, *serial* ones to sequential flows, *parallel* ones to AND-Gateways.

Example 7. *Figure 4 illustrates the algorithm for the sequential arrangement. The algorithm starts with the initial list $\mathcal{T} = \{B\}$ of necessary tasks in the process model and the sequential constraints $\mathcal{S}$. Next we extend $\mathcal{T}$ to all tasks that are consequences of an antecedent task in $\mathcal{T}$. This results in the new list $\mathcal{T} = \{A, B, C, D\}$, see Figure 4(a). E is not in $\mathcal{T}$ because E always is an antecedent in the pattern. Next, we generate the dependency graph using $\mathcal{T}/\mathcal{S}$ and compute the transitive closure. Figure 4(b) shows the result. We then compute the modules of the graph, see Figure 4(c). Lastly, we transform the modular decomposition tree into a process tree, see Figure 4(d).*

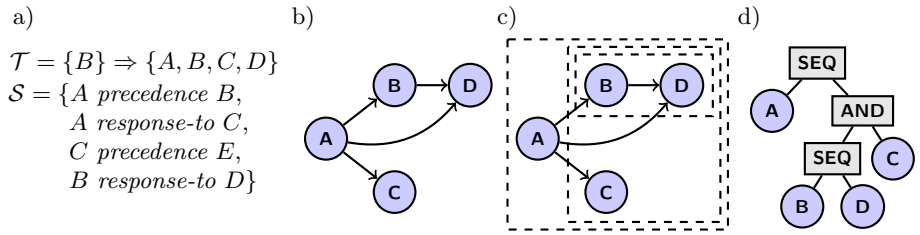


Fig. 4. Extend the List of Required Tasks (a), Construct a Dependency Graph (b), Perform Modular Decomposition (c), Transform to Process Tree (d).

The transformation from the process tree to a modeling language is straightforward using a template for each control-flow element supported. Figure 5 illustrates this for BPMN, Petri Nets and WS-BPEL.

4 Applying Parallel Constraints with Optimization

In addition to sequential tasks, parallel constraints exist. To cope with these constraints, we apply a post-processing scheme to the process just generated.

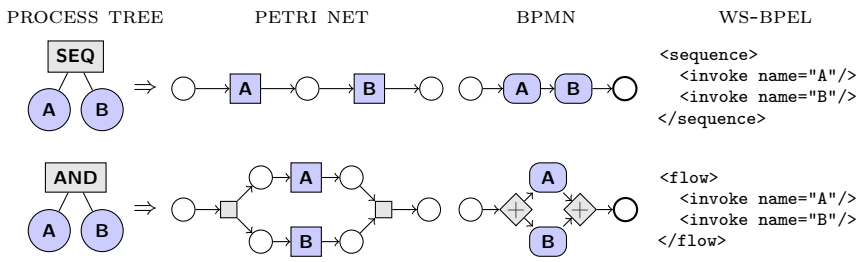


Fig. 5. Transformation of a Process Tree to Petri Net, BPMN or WS-BPEL.

At first sight, instead of our algorithm in Section 3, it would also be possible to combine this post-processing scheme with other synthesis approaches for sequential constraints, e. g., [2], [4], [1]. However, a comparison of ours (see [14]) has shown that these competitors do not scale well with the size of the use case. Hence, we do not study the combination with these synthesis approaches in what follows. Our post-processing scheme schedules the tasks with fewer parallel paths, so that the parallel constraints are met. Because reducing the degree of parallelism affects the optimization criterion, e. g., processing time, we need to observe it as well. The core idea behind our approach is to iteratively merge two parallel lanes into one, until the constraints are fulfilled. In each iteration we look at each candidate pair of parallel lanes and rate it according to the effects it would have. For instance, we could look at the reduction of *resource consumption* versus the increase in *processing time*. We then apply the candidate with the highest rating. We repeat this until all constraints are fulfilled. In the worst case, this means arranging all tasks in a sequence. We do not automatically select the candidate with the best rate. This is because this greedy approach might not lead to a globally optimal result. See Example 8. Instead we choose the candidate according to a probability function and repeat this selection several times, with the chance to achieve a globally optimal result.

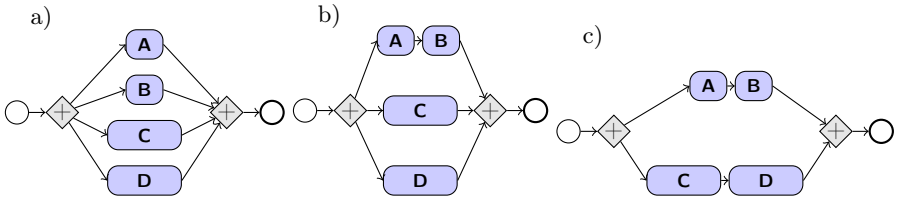


Fig. 6. A Greedy Approach Might Not Lead to a Globally Optimal Result.

Example 8. *Figure 6(a) shows a process model with four tasks* $\{A, B, C, D\}$. *Task* A *and* B *take* $1s$ *to complete, tasks* C *and* D *take* $2s$. *We want to reduce the model*

to use a maximum of two parallel lanes. A greedy approach would combine A and B in the first step; see Figure 6(b). This would lead to a suboptimal solution in the next iteration, see Figure 6(c), because combining C and D leads to an execution time of 4s, but combining C with A and D with B would lead to 3s for all paths.

Algorithm 2 ParallelConstraints (ProcessTree *original*, Parallel Constraints $\mathcal{P}$, Additional Constraints $\mathcal{C}$): ProcessTree *best*

```
 1: for it = 0 until it = ITERATIONS do
 2:     ProcessTree tmp ← original.copy
 3:     while ∃p ∈ P that is not fulfilled do
 4:         Compute all candidates C of parallel lanes
 5:         chosen ← HeuristicSelection(C) // See Algorithm 3
 6:         ProcessTree tmp ← combine the two lanes of chosen in tmp
 7:     end while
 8:     if ∃c ∈ C that is not fulfilled then
 9:         continue
10:     end if
11:     if tmp has shorter processing time than best then
12:         best ← tmp
13:     end if
14: end for
15: return best
```

Algorithm 2 is our post-processing scheme for purely sequential process models that helps in fulfilling parallel constraints. Each iteration collects candidates $\mathcal{C}$ (Line 4). A candidate $c \in \mathcal{C}$ is a pair of two parallel lanes on any hierarchy level of the process model. In each iteration, our algorithm chooses a candidate for a merging using a heuristic. Algorithm 3 will be the algorithm to select a candidate using the gain function and the probability function $p(c)$. Algorithm 2 receives the candidate c and combines its lanes c (Line 6). We iterate until the resource constraint is fulfilled (Lines 3–7). If the process tree found violates a constraint on the execution as a whole we omit the candidate (Line 8–10). For instance, in our evaluation scenario the complete processing time of the process execution must not exceed a limit L. Line 11 checks if the solution found (*tmp*) has a shorter overall processing time than the best one so far. We repeat Algorithm 2 until the maximal number of iterations is reached (Lines 1–14). Other abort criteria, e. g., maximum computation time, are conceivable as well. We finally return the best solution found for the resource constraint (Line 15).

Next, we explain which candidate we choose in each iteration, i. e., Algorithm 3. A candidate c is good if its deparallelization helps fulfilling all constraints without lowering the quality of the process model significantly, e. g., total processing time. To this end, we define the gain of a candidate. A gain function g is a function with the following characteristics:

– $\forall c \in \mathcal{C} : g(c) \geq 0$
– If a Candidate c_1 is a better candidate for deparallelization compared to a Candidate c_2 then $g(c_1) > g(c_2)$.

The concrete function quantifying the gain depends on the scenario. In our scenario for instance, we consider parallel constraints as well as the total runtime of the process instances and the waiting time of a human worker. Example 9 motivates and introduces the gain function we use there.

Example 9. *To quantify the gain, the reduced degree of parallelism of the resource $\Delta_{res}(c)$ when we combine the two lanes in c is important. Additionally, we calculate how the deparallelization of c would extend the processing time $\Delta_{run}(c)$ and the human worker time $\Delta_{hum}(c)$. If the combination of the two lanes brings the process model under the limit for the resource, we set $\Delta_{res}(c)$ to the limit of the resource. The rationale behind this design choice is that a reduction under the limit does not yield any performance gain. For instance, if we assume that the process model before the reduction uses 5 resources, and the limit of the resources is 4, then $\Delta_{res}(c)$ is at most 1. Next, we calculate the gain of the candidate as*

$$g(c) = \frac{\Delta_{res}(c) + \lambda}{\sigma \cdot \Delta_{hum}(c) + \Delta_{run}(c) + \lambda}$$

σ with $\sigma > 1$ is a constant that ensures that the human processing time is deemed more important than the total processing time. λ is a constant with $\lambda > 0$ to trade off quality vs. performance. A larger λ leads to a broader search with a potentially better result but a longer runtime. To illustrate, a large λ dominates the term and renders the selection less dependent on the other addends. This in turn would cause the algorithm to converge slower, but tends to prevent the algorithm from getting trapped in a local minimum. See Section 5.2 for experiments on the influence of λ.

Our goal is to select candidates with a high gain to reduce the resource consumption efficiently. Our algorithm combines the two parallel lanes of the candidate into one. The algorithm then repeats the selection and combination steps, until all constraints are fulfilled. Observe that the selection of a candidate influences the *gain* values of subsequent candidates. The selection of the best candidate in one iteration could lead to an end result that is not optimal, cf. Example 8. Hence, we add some randomness to the selection of the candidates and repeat it several times. A completely random selection could lead to a long runtime of our algorithm. We resolve this by weighting the probability of the candidates $p(c)$ with their gain. To make the selection simple, we require that $\sum_{c \in \mathcal{C}} p(c) = 1$, i. e., $p(c)$ is a probability mass function. Thus, for all candidates, $p(c) \in [0, 1]$. The probability should be higher for candidates with a high gain. We choose the function $p(c)$ as follows:

$$p(c) = \frac{g(c) - \text{MIN} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)}$$

We see that $p(c) \in [0,1]$ for all candidates. Obviously, MIN is the lowest value for $g(c)$. ϵ is a small constant to avoid a division by zero.

Lemma 2.

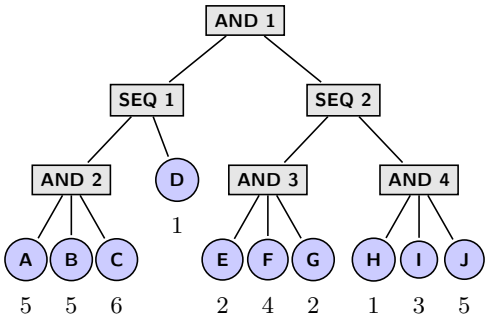
$$p(c) = \frac{g(c) - \text{MIN} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)} \qquad \textit{is a probability mass function.}$$

Proof. To show the lemma, we need to prove that $\sum_{c \in \mathcal{C}} p(c) = 1$.

$$\begin{aligned}\sum_{c \in \mathcal{C}} p(c) &= \sum_{c \in \mathcal{C}} \frac{g(c) - \text{MIN} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)} = \frac{\sum_{c \in \mathcal{C}} g(c) - \text{MIN} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)} \\ &= \frac{\sum_{c \in \mathcal{C}} g(c) + |\mathcal{C}|(-\text{MIN} + \epsilon)}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)} = \frac{\sum_{c \in \mathcal{C}} g(c) - |\mathcal{C}|(\text{MIN} - \epsilon)}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)} = 1\end{aligned}$$

□

Example 10 lists the function p for a simple example.



Candidate	g(c)	p(c)
(seq 1,seq 3)	0.18	0.051
(A,B)	0.27	0.080
(A,C)	0.22	0.064
(B,C)	0.22	0.064
(E,F)	0.05	0.010
(E,G)	1.00	0.313
(F,G)	0.05	0.010
(H,I)	1.00	0.313
(H,J)	0.09	0.022
(I,J)	0.03	0.003

Fig. 7. A Process Tree with its Candidates for the Reduction and their Respective Gain $g(c)$ and Probability $p(c)$

Example 10. *Figure 7 shows a process tree. The numbers under the tasks are their respective runtimes. The table on the right hand side shows the gain* $g(c)$ *for* $\lambda = 0.1$*. The sum of all gains is* $\sum_{c' \in \mathcal{C}} g(c') = 3.33$*, the minimum is* 0.03 *and* $|\mathcal{C}| = 10$*. We have set* ϵ *to* 0.01 *to get the probabilities* $p(c)$ *in the table on the right hand side.*

5 Evaluation

Subsection 5.1 elaborates on our evaluation scenario, the commissioning of vehicles. Subsection 5.2 describes experiments regarding the performance of our algorithm and the role of its parameters. Subsection 5.3 describes results with our use case.

Algorithm 3 HeuristicSelection (Candidates $\mathcal{C}$): candidate *best*

for all $c \in \mathcal{C}$ **do**
  Calculate $g(c)$
end for
$\text{MIN} \leftarrow \min(g(c) \mid c \in \mathcal{C})$
for all $c \in \mathcal{C}$ **do**

$$p(c) = \frac{g(c) - \text{MIN} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\text{MIN} - \epsilon)}$$

end for
$rand \leftarrow$ Random Number between 0 and 1
$last \leftarrow 0$
for all $c \in \mathcal{C}$ **do**
  if $rand > last \vee rand \leq last + p(c)$ **then**
    $best \leftarrow c$
    break
  else
    $last \leftarrow last + p(c)$
  end if
end for
return *best*

5.1 Scenario Commissioning of Vehicles

Commissioning consists of two phases: first, testing if all parts of the vehicle function properly; second, installment of software on the control units (ECUs) built into the vehicle. To this end, each vehicle is moved to several testing stations. At each station, a factory worker connects the vehicle to a diagnostic system, i. e., a workflow system for commissioning [16], using a mobile testing station (MPS, an acronym for the German word Mobile Prüfstation). The MPS invokes several operations on the vehicle and presents tasks to the worker via a hand-held terminal. ECUs are components built into the vehicle which control certain features of the car. E. g., the ECU MOT controls the engine electronics. Commissioning processes are inherently complex. Typically, there are hundreds of tasks for each vehicle. For instance our evaluation use case consists of 496 tasks. The tasks are arranged in up to 14 parallel lanes, the maximal capacity of the bus network. Each testing station and respective process model uses a preplanned number of production cycles, i.e., sets itself a time limit. The execution of the model needs to be complete within this time or will cause a major disturbance of the production flow.

The current situation is that a dedicated department plans and implements those commissioning process models by hand, using specific authoring tools. There are several factors rendering process-model design complex and expensive. These include the increase of the number of assisting and media components in the vehicle to be tested, or shorter production cycles for the commissioning. This combination leads to a constant increase in the number and complexity of process

models for the commissioning of a vehicle project. At the same time, for each process model several hundreds of constraints exist. For instance, the majority of operations require other operations as pre- or postcondition, and the resources used by operations have limited capacity. Observe that loops are unnatural in commissioning processes, since a feature is tested only once. If a problem occurs and is fixed, a new commissioning process is started later on. The rationale is that vehicle production uses a cycle system, i. e., each production step has a fixed time limit for its completion. Iterations with an unclear number of repetitions in a process model make predictions of its runtime overly difficult.

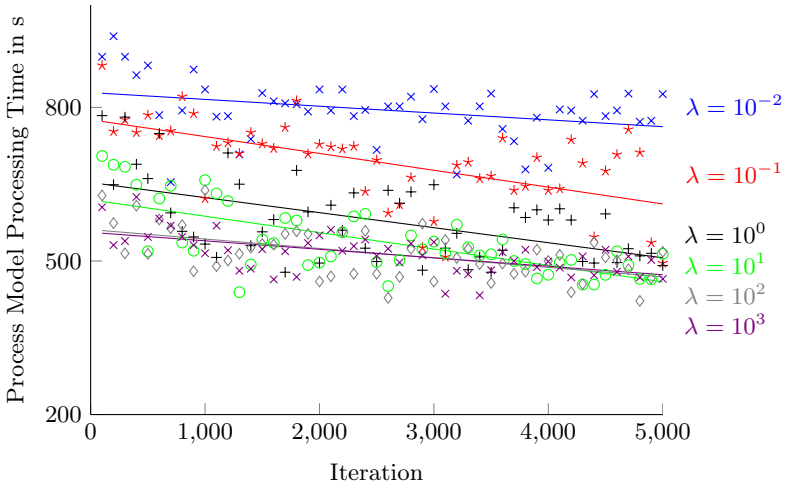


Fig. 8. Processing Time as a Function of the Number of Iterations and of λ

5.2 Performance Evaluation

In a first step, we have tested the runtime and performance of our algorithms by means of simulations. To this end, we have synthesized process models using our approach. The specification for the synthesis, e. g., tasks and structure, were similar to what we have found in industrial processes. More specifically, the number of tasks has been 447 and the average outdegree of each node has been 13.66. For our evaluation, we have specified 48 patterns of constraints that have resulted in 465 sequential constraints. As parallel constraints we define that each component should only be accessed once and that the maximal degree of parallelization should not exceed 5. In total, this results in 78 parallel constraints.

Figure 8 shows the processing time as a function of the number of iterations of our algorithm and of the value of λ. It turns out that the processing time decreases with the number of iterations and does depend on λ. For $\lambda < 1$, the quality of the solutions decreases significantly, while for $\lambda > 10^2$ the quality increases only moderately. Figure 9 shows the calculation time, i. e., the time that

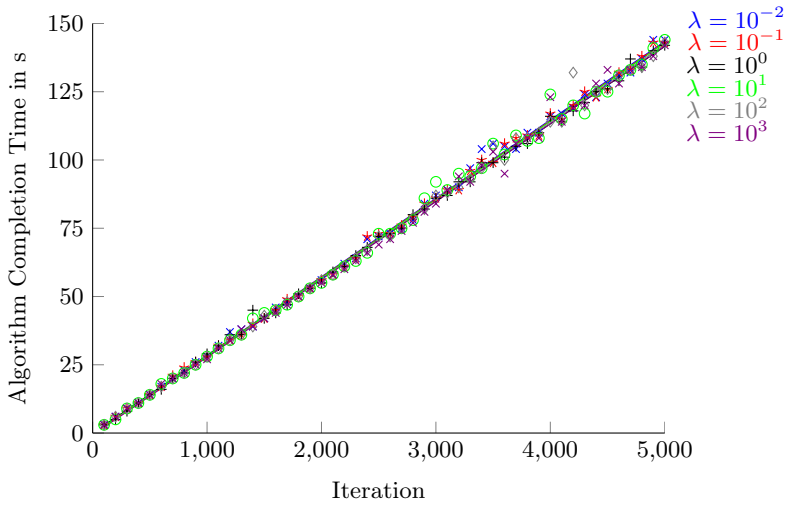


Fig. 9. Calculation Time as a Function of the Number of Iterations and of λ

our algorithm needs to compute the result. It is clearly linear with the number of iterations. A larger value for λ decreases the computation time, but only slightly. Given these results we have chosen $\lambda = 10^2$ for our real-world evaluation scenario.

5.3 Evaluation with Industrial Commissioning Processes

To study whether our synthesis approach achieves results comparable to manually designed ones we have compared our synthesized process models to real ones from industry. More specifically, we have synthesized models for exactly the same vehicle project and tasks. We had manually crafted models for as reference points.

Our approach focuses on the process models for a new vehicle series. The process model is large in size, i. e., consist of 496 tasks. 96 human tasks exist. We have used the algorithms of Section 3 and Section 4 for the synthesis with $\lambda = 10^2$ and $\epsilon = 0.1$. To compare the performance of the synthesized process model to the manually-designed one objectively, we simulate the actual commissioning with both process models with real-world runtime data of the various tasks. In total we used 193 instances of the process model. The instances can have one of two configurations $C1$ (140 instances) or $C2$ (53 instances).

Figure 10 shows the runtimes of those instances for the manually designed and the synthesized process models. For $C1$, there is an average reduction of 5.92 s or 5.5 %. For $C2$ the average reduction is 209.13 s or 50.39 %, compared to the manually designed process model. The smaller reduction in the $C1$ configuration is because there is a critical path consisting mainly of human tasks, i. e., only minor optimization is possible.

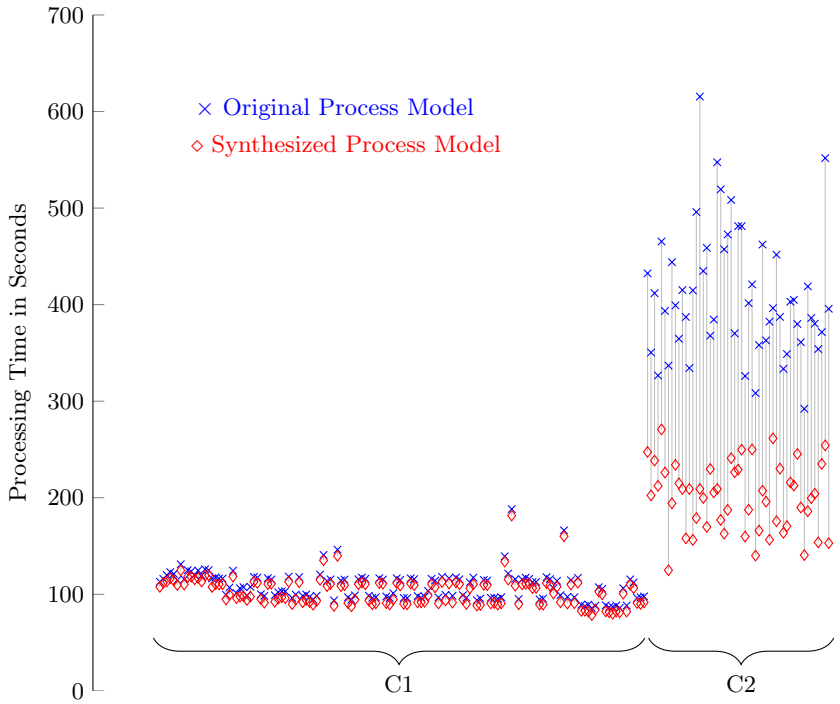


Fig. 10. Processing Time of the Original and the Synthesized Process Model

6 Related Work

A schedule is the planned execution of a set of activities. The RCSP (Resource Constraint Scheduling Problem) [17] is an optimization problem to find an optimal execution for a given set of activities, i. e., a function s that maps each task $t \in T$ to a starting time. In general, scheduling algorithms can take both predecessor relations between activities and capacity limits of resources into account. The reasons why an activity A is scheduled before an activity B are manifold, e. g., to comply with a given constraint, or because of optimization. In consequence, the process models are inherently inflexible and over-specified. It is therefore not possible to detect parallel lanes in a schedule in contrast to our approach.

We now discuss concrete approaches that are related, before stating why all of them do not solve our problem. [18] 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. [18] determines relationships between the tasks of a process model and generates an org using these relationships. Next, [18] decomposes the org into a Modular Decomposition Tree. In contrast to our approach, [18] 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, several process models are possible. Awad et al [1] propose an approach to synthesize a process model from compliance rules in LTL notation.

They first extract all valid executions paths for the compliance rules and build a model of the correct traces. Next they synthesize a process model from the model using an approach similar to process mining [19]. Yu et al [2] synthesize WS-BPEL process models from PROPOLS [20] patterns. PROPOLS is a language for a set of property-specification patterns that abstract from the temporal specification, similarly to the patterns of Dwyer et al [11]. PROPOLS pattern can be expressed as finite state machines. The algorithm of [2] combines the patterns to one state machine and extracts all accepting paths of the state machine. Next, an algorithm similar to the α-algorithm [21] for process discovery is used to extract a process model. Both [1] and [2] require to extract all paths from the specification. This is hardly possible for parallel constraints. This is because the number of paths typically grows exponentially with the degree of parallelization. Additionally, [1] and [2] only consider the case that a unique process model for the specification exists.

7 Conclusions

We have presented an approach for the synthesis of process models from both sequential and parallel constraints. To this end, we use a two-step approach. First, we generate a model fulfilling the sequential constraints and then a novel algorithm that deparallelizes the process to fulfill the parallel constraints as well as additional optimization criteria. We have evaluated our approach using the real-world use case of commissioning in vehicle manufacturing. An important takeaway is that our synthesized models are significantly better than these reference points. We conclude that our approach can indeed deal with large specifications occurring in real scenarios.

References

[1] Ahmed Awad et al. "An Iterative Approach for Business Process Template Synthesis from Compliance Rules". In: *Advanced Information Systems Engineering*. LNCS 6741. Springer Berlin Heidelberg, 2011, pp. 406–421.

[2] Jian Yu et al. "Synthesizing Service Composition Models on the Basis of Temporal Business Rules". en. In: *Journal of Computer Science and Technology* 23.6 (Nov. 2008), pp. 885–894.

[3] Richard Mrasek, Jutta Mülle, and Klemens Böhm. "Automatic Generation of Optimized Process Models from Declarative Specifications". In: *Advanced Information Systems Eng.* Springer Berlin Heidelberg, 2015, pp. 382–397.

[4] Niels Lohmann. "Compliance by design for artifact-centric business processes". In: *Information Systems*. Special section on BPM 2011 conference 38.4 (June 2013), pp. 606–618.

[5] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. "Semantics and analysis of business process models in BPMN". In: *Information and Software Technology* 50.12 (2008), pp. 1281–1294.

[6] Richard Mrasek, Jutta Mülle, and Klemens Böhm. "A new verification technique for large processes based on identification of relevant tasks". In: *Information Systems* (2014).

[7] Stefanie Rinderle-Ma, Ly Linh Thao, and Peter Dadam. "Business process compliance". In: *2008 EMISA Forum*. 2008, pp. 24–29.

[8] Richard Mrasek et al. "User-Friendly Property Specification and Process Verification - a Case Study with Vehicle-Commissioning Processes". In: *BPM Conf*. Springer Berlin Heidelberg, 2014, pp. 326–341.

[9] Ahmed Awad, Gero Decker, and Mathias Weske. "Efficient Compliance Checking Using BPMN-Q and Temporal Logic". In: *Business Process Management*. LNCS 5240. Springer Berlin Heidelberg, 2008, pp. 326–341.

[10] Linh Thao Ly et al. "SeaFlows Toolset – Compliance Verification Made Easy for Process-Aware Information Systems". In: *Information Systems Evolution*. LNBIP 72. Springer Berlin Heidelberg, 2011, pp. 76–91.

[11] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in property specifications for finite-state verification". In: *International Conference on Software Engineering*. 1999, pp. 411–420.

[12] James P. Womack, Daniel T. Jones, Daniel Roos, and Massachusetts Institute of Technology. *Machine that Changed the World*. en. London: Free Press, 2007.

[13] Maja Pešić, Helen Schonenberg, and Wil M. P. van der Aalst. "DECLARE: Full Support for Loosely-Structured Processes". In: *11th IEEE Intl. EDOC 2007*. 2007, pp. 287–287.

[14] Richard Mrasek, Jutta Mülle, and Klemens Böhm. *Automatic Generation of Optimized Process Models from Declarative Specifications*. en. Technical Report 2014-15. Karlsruhe: KIT Scientific Publishing, Nov. 2014.

[15] Ross M. McConnell and Fabien de Montgolfier. "Linear-time modular decomposition of directed graphs". In: *Discrete Applied Mathematics* 145.2 (2005), pp. 198–209.

[16] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur*. 2011.

[17] Jacek Błażewicz, Jan Karel Lenstra, and Alexander H. G. Rinnooy Kan. "Scheduling subject to resource constraints: classification and complexity". In: *Discrete Applied Mathematics* 5.1 (Jan. 1983), pp. 11–24.

[18] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. "Structuring acyclic process models". In: *Information Systems*. BPM 2010 37.6 (Sept. 2012), pp. 518–538.

[19] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. 1st. Springer Berlin Heidelberg, 2011.

[20] J. Yu et al. "Pattern Based Property Specification and Verification for Service Composition". In: *Web Information Systems – WISE 2006*. LNCS 4255. Springer Berlin Heidelberg, 2006, pp. 156–168.

[21] Wil M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster. "Workflow mining: discovering process models from event logs". In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (Sept. 2004), pp. 1128–1142.